Dynamic Loading and Class Management in a Distributed Actor System

by

Gilles Carlo

Project Report submitted to the faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science

APPROVED:

[illegible]

Prof. Dennis Kafura, Chairman

[illegible]

Prof. James D. Arthur

[illegible]

Prof. Scott F. Midkiff

July, 1993

Blacksburg, Virginia

LD
5655
V851
1993
C374
c.2

c.2

Dynamic Loading and Class Management in a Distributed Actor System

by

Gilles Carlo

Committee Chairman: Prof. Dennis Kafura

Department of Computer Science

(ABSTRACT)

The goal of this project was to develop part of an environment that would allow the creation of distributed applications using the actor model in an heterogeneous environment. The actor model is realized by ACT++, a C++ framework for building actor applications.

This project is concerned only with the problems of the creation, destruction and the invocations of the methods of a remote server actor. A related project concerns the client activities and the message transfer. Three elements comprised the solution: a run-time loading entity using the facilities of a dynamic linker called "Dld," a query service to identify the classes present in an object file using a tool derived from the source-level debugger "GDB" and a directory service allowing both classes and actors to be located in memory. The solution was tested on several simple examples.

The fundamental features of C++ and of the actor model have been retained in the distributed environment. The typing mechanisms used by C++ are preserved, and both polymorphic and overloaded functions are available. Regarding the actor model, the main components, namely actors, behaviors, messages and the replacement behavior are present. However, a choice had to be made concerning the communication model and the argument passing semantics. C++ and the actor model support synchronous and asynchronous communications, respectively. The latter was chosen, as our solution was based on the actor model and its message passing mechanism. C++ allows by reference parameters, while in the distributed environment, only by value parameters are allowed.

It appears that the tools derived from GDB and Dld could also be used as a software engineering tool, allowing the dynamic linking and unlinking of a class, the creation of objects of that class, and the invocation of its methods for testing purposes.

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dennis Kafura, for his help and guidance in the realization of this project.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1
INTRODUCTION

With computing environments becoming increasingly distributed and heterogeneous, the creation of an environment supporting both machine and language heterogeneity would decrease the cost of the development of distributed applications, and also increase the interoperability of systems. The use of the object-oriented paradigm can help in creating such an environment. The uniform interface and the property of data encapsulation provided by objects offer a uniform framework for communication between objects and the possibility to hide implementation details from the user.

The actor model is a conceptual model for concurrent object-oriented programming, based on the message passing paradigm. This model has features which closely correspond to several important aspects of distributed systems. First, actors represent autonomous objects that can execute concurrently. Second, the message passing mechanism used by the actor model is inherent in distributed systems.

This project is an attempt to create part of an environment that would support the distribution of an actor-based system in an heterogeneous environment. The environment created supports object interaction on the server side. Namely, it provides a run-time loading entity, a query service to identify the classes present in an object file, and two directory services allowing both classes and actors to be located in memory. A related project concerns the client activities and the message transfer.

In the remainder of this report, Chapter 2 presents background on distributed object-based programming systems. Important concepts are defined, followed by a discussion on the advantages of object-based systems for developing distributed applications. In Chapter 3, the actor model is outlined. An implementation of this model, ACT++, is also described. Chapter 4 presents the environment created, the different components involved and their implementation. Finally, Chapter 5 discusses the capabilities and limitations of the environment created. Some enhancements are also suggested for future work.

Chapter 2

DISTRIBUTED OBJECT-BASED PROGRAMMING SYSTEMS

As computer usage is spreading into all functions of companies, and with the request for integrated products, software applications are becoming increasingly sophisticated in terms of size and complexity. Consequently, there is a growing need for new paradigms that ease the development, maintenance and ensure a high quality level. Object-oriented systems have emerged as one of the most promising paradigms for the design, construction and maintenance of large scale systems.

The next section will present the different definitions useful to describe this paradigm.

2.1 Objects and Object-Based Programming Languages

Peter Wegner presents in [Weg87] different properties that allow one to analyze and categorize object-based languages depending on the set of properties that they provide. The first property described are *objects*.

Object: An object has a set of "operations" and a "state" that remembers the effect of operations.

An *object-based language* is then a language which supports objects as a feature.

The second notion presented is that of class which can be defined as follows:

Class: A class is a template (cookie cutter) from which objects may be created by "create" or "new" operations. Objects of the same class have common operations and therefore uniform behavior. Classes have one or more "interfaces" that specify the operations accessible to clients through that interface. A "class body" specifies code for implementing operations in the class interface.

A language that offers the objects and classes features is then called a *class-based language.*

The third property consists of providing a mechanism to build a hierarchy of the previously defined classes. This mechanism, called inheritance, is defined as follows:

Inheritance: A class may inherit operations from existing "superclasses" and may have its operations inherited by as yet undefined "subclasses." Inheritance from a single superclass is called single inheritance; inheritance from multiple superclasses is called multiple inheritance.

A language containing those three features is then called *object-oriented.* The previous definitions can be summed up by the Figure 2.1.

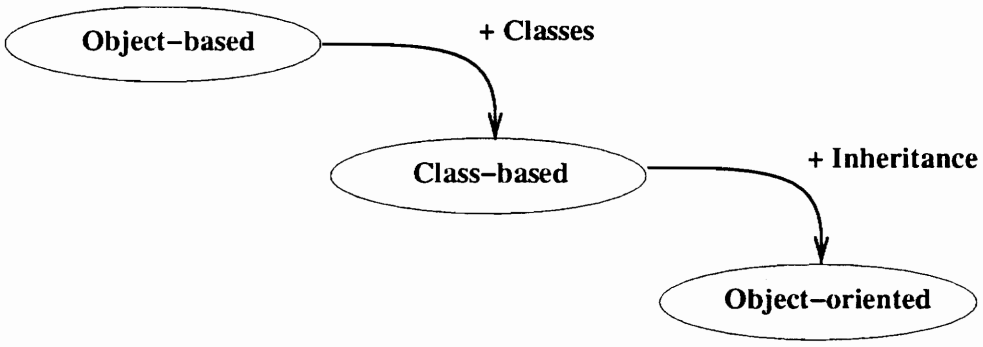


Figure 2.1: Wegner's Classification Scheme.

Wegner then defines two other properties, data abstraction and strong typing, that define subclasses of the three different language categories. Their definition is given below.

Data abstraction: A data abstraction is an object whose state is accessible only through its operations.

Strong typing: A language is strongly typed if type compatibility of all expressions representing values can be determined from the static program representation at compile time.

Object-oriented languages which have strong typing and require all objects to be data abstractions are called *strongly typed object-oriented languages.*

Wegner's definition of object-related systems has gained wide-acceptance. However, as noticed in [BGM89], this definition may be considered as too restrictive. The drawback of this definition is that it is sometimes based on mechanisms rather than on properties. In particular, some languages are not considered object-oriented, even though they exhibit features comparable to inheritance. For example, some languages provide a mechanism called delegation. Delegation is defined by Wegner as follows.

Delegation: Delegation is a mechanism that allows objects to delegate responsibility for performing an operation, or finding a value, to one or more designated "ancestors".

Inheritance can be viewed as a specialization of delegation in which the entities that inherit are classes. However, Stein in [Ste87], demonstrates that delegation is inheritance. In [BGM89], inheritance and delegation are viewed as mechanisms to implement a property called inclusion polymorphism. Blair et al. [BGM89] in fact give an alternative definition of object-oriented systems, based on the notion of polymorphism. They first define set-based abstraction as follows:

Set-based abstraction In set-based abstraction, all entities in the problem domain are considered to belong to sets. Sets abstract out common properties and behaviors of a group of objects. It is possible for a single entity to belong to more than one set through intersection. Similarly, sets can themselves be subsets of more general sets.

Now, the definition of inclusion polymorphism can be given as follows.

Inclusion polymorphism: Inclusion polymorphism implies that objects can belong to more than one set (through set intersection). Hence, operations applicable to one set may be applicable to objects of another set through inclusion.

Figure 2.2 illustrates the relationship between inheritance and inclusion polymorphism. An object which is an instance of one class can be used as if it is an instance of another class provided that the first class has a subclass relationship with the second class.

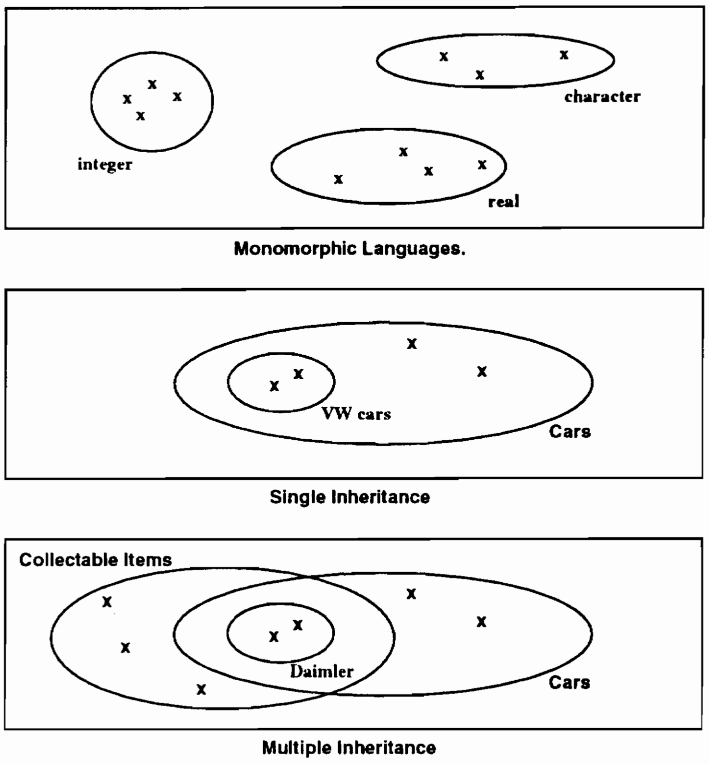


Figure 2.2: Relationship between inclusion polymorphism and inheritance.

Delegation supports inclusion polymorphism also. One object can be used as if it is another object if it delegates operations to that object.

Another kind of polymorphism is the operation polymorphism. This notion refers to the possibility of having methods with the same name that can be applied to different objects which have no relationship in terms of inclusion. Such methods are said to be *overloaded.*

Finally, some systems provide *virtual methods* (or functions), and *abstract classes.* A virtual method is a method which is named and specified in a class and which can be redefined in the subclasses of the class in which it is defined. Actually, a virtual method may not be implemented in the base class, or in some of its descendants. When the function is declared but not implemented, the function is a *pure virtual function.* A class containing a pure virtual function is called an abstract class. An abstract class can be instantiated only as an instance of a subclass for which virtual resources of the class are implemented.

2.2 Software Engineering Principles Supported by Object-Oriented Systems

A usual human approach to solve a complex problem is to divide it into less complex subproblems, and to adopt a top-down decomposition approach. The emphasis is first on the study of the high level components and their interactions, and the implementation details are initially of little or no concern.

The design of an object-oriented application is a process similar to the one previously described. The problem is first modeled as a set of multiple, interacting, autonomous entities and the relationships among them. Those entities and relationships are then assembled to form the basic architecture of the application. The idea of dividing a problem into subproblems is not new. This is also the approach taken in the development of an application in a procedural language. However, the main difference is that the analysis and design stages of the software life cycle are, in fact, only one stage when an object oriented language is used, while the analysis stage just provides input for the design stage in the case of a procedural language. Because the abstractions used by the programmer in the analysis of the problem domain are provided by the language, the development of the model of the problem becomes an integral part of the design. In fact, the object-oriented paradigm supports the complete software development life cycle:

- Analysis

- Design
- Implementation

As those three stages rely on the same model, there is no need for a translation from one phase to another, which is usually time consuming and error prone.

In addition, object-oriented programming languages emphasize a style of programming that follows traditional software engineering principles. According to these principles, a program should have the following five characteristics:

Abstract. The design concepts should be separated from the implementation details. The object-oriented paradigm offers three types of abstraction.

- The object classes provide *data abstraction* by encapsulating a data structure along with the operations to be performed on it. This allows the designer to think at a higher level with objects, and only later be concerned about the details of how the data structures and operations are to be implemented in a stepwise refinement approach. Also, even if a product is drastically changed, it is unlikely that the object interface and instance variables will be greatly modified.
- *Procedural abstraction* is allowed by extending the language with the user defined operations. All the methods defined can be used like predefined procedures.
- The object-oriented paradigm also provides *iteration abstraction* by allowing the designer to specify a loop at a higher level, and only later be concerned with the details of how that loop is to be implemented.

Structured. A large program should be decomposed into components of a manageable size, with well defined relationships established between the components. This property is very naturally achieved by the use of object classes and inheritance.

Modular. The internal design of each component should be localized so that it does not depend on the internal design of any other component.

In an object-oriented application, the classes are the modules. This means that not only the design process supports modularity, but the implementation process supports it as well through the class definition. As a consequence, the object-oriented paradigm provides natural support for decomposing a system into modules.

Object classes lead to a weak coupling (few connections) between classes. The classes are designed as a collection of data and the set of allowable operations on that set of data. Therefore, the interface operators of a class are inward-looking in the sense that they are intended to access or modify only the internal data of that class.

Object classes provide also a high level of cohesion: first, because it is a model of some entity; second, because it offers informational cohesion. A module has *informational cohesion* if it performs a number of functions, each with its own entry point, with independent code for each function (the methods), all performed on the same data structure (the attributes).

Concise. The code should be clear and understandable. This can be achieved with the use of classes, as they represent a real object definition, or define a user abstraction.

Verifiable. The program should be easy to test and debug. It is very easy to test and debug a class separately from the other classes. As classes provide a high cohesion and a low coupling, almost no testing is required after the integration phase.

The object-oriented paradigm offers another great advantage in terms of *maintainability* and *reusability*. This is primarily due to two reasons:

- Data abstraction.
- Inheritance.

A great part of maintenance deals with adding functionalities to the application software. The inheritance mechanism supports extending design in two ways.

First, the inheritance relation facilitates the reuse of existing definitions to ease the development of new definitions. It reduces the code bulk by reducing the need to redevelop common functionalities, and as a consequence, increases the consistency.

Second, objects become polymorphic with the use of inheritance. Polymorphism is the ability to take more than one form. In object-oriented programming, it refers to the ability of an object to refer at run-time to instances of various classes. For example, a golf, which is an instance of the class VW cars can also be seen as an instance of the superclass cars (see Figure 2.2). Another feature called *dynamic binding* is needed to use effectively the polymorphism property of objects. Dynamic binding means that the code associated with a given procedure call is not known until the moment of the call at run-time. This is very powerful, as it allows procedures or functions to operate on more than one type.

The maintenance is also easier because the narrow, clearly defined interface of a class helps in the integration process with the other components.

Concerning reusability, the object-oriented paradigm combines design techniques and language features to provide strong support for the reuse of software modules. Every time an instance of a class is created, reuse occurs. The development of meaningful abstractions also encourages reuse.

2.3 Distributed Object-Based Programming Systems

A distributed, object-based programming system provides the features of an object-based programming system as well as a distributed computing environment.

According to Chin and Chanson [CC91], a distributed object-based programming system should typically have the following characteristics:

Distribution. A distributed object-based programming system combines a network of independent, possibly heterogeneous, workstations.

Transparency. The system may hide the details of the distributed environment. For example, the user should not have to know the physical location of an object to be able to make an invocation on it. This feature is called *location transparency.* The user should not have to worry whether the definition of an object is in memory or in secondary storage.

Data Integrity. A system may provide *persistent objects*, which means objects that outlast the life of an application. In this case, the system must ensure that an object is always in a state that is the result of the successful termination of an operation. If a method invocation does not complete successfully, the changes to the object should be undone to restore the object in its initial state.

Fault Tolerance The failure of a workstation should not induce the failure of the whole system. The remainder of the system should be able to offer the services that were not dependent on the workstation that has failed.

Availability. This property means that the failure of one part of the system does not affect the number of services offered. If the failure is so basic that it is not possible to process vital requests, the system may be shut down, in order to allow a restoration of full service later on.

Recoverability. If a workstation fails, the system should restore the persistent objects that resided on it.

Object Autonomy. The system may permit the owner of an object to specify the clients that have the authority to make invocations on the object.

Program Concurrency. The system should be able to assign the objects of a program to multiple processors so they may execute concurrently.

Object Concurrency. An object should be able to serve multiple, nonmodifying invocation requests concurrently. Note that true concurrency is possible only if the object resides in a multiprocessor, since only one request can be processed at any one time otherwise.

Improved Performance. A well-designed program should execute more quickly than in a conventional system.

2.4 Passive Object-Model versus Active Object-Model

The *composition* of objects is the relationship between processes and objects. Two approaches can be taken.

The Passive Object Model: In this model, processes and objects are two separate entities. A process is not bound, nor restricted to one particular object. Instead, a single process is used to perform all the operations required to satisfy an action.

The Active Object Model: Processes are bound and restricted to one particular object. Several processes may be created for and assigned to each object to handle its invocation requests. In this approach, many processes may be involved in performing a single action. Two variations exist for this model. In the dynamic variation, as many processes as needed may be created per object. In the static variation, a fixed number of processes are created for each object.

Figure 2.3 illustrates how an action is performed in both models. This action involves the invocation of three methods, each belonging to a different object. In the passive object model, when a process makes an invocation on another object, its execution in the object in which it currently resides is temporarily suspended. The process is then mapped into the address space of the second object, and executes the method invoked. When the execution of the method is completed, the process returns to the original object and resumes the execution of the original operation. In the active object model, when a process makes an invocation on another object, it sends an invocation request along with the appropriate arguments to that particular object and waits for a result. A server process belonging to the object which method is being invoked, accepts the request and executes the method. When the execution of the method is completed, the server process returns a result to the calling process. The calling process then resumes the execution of the original operation.

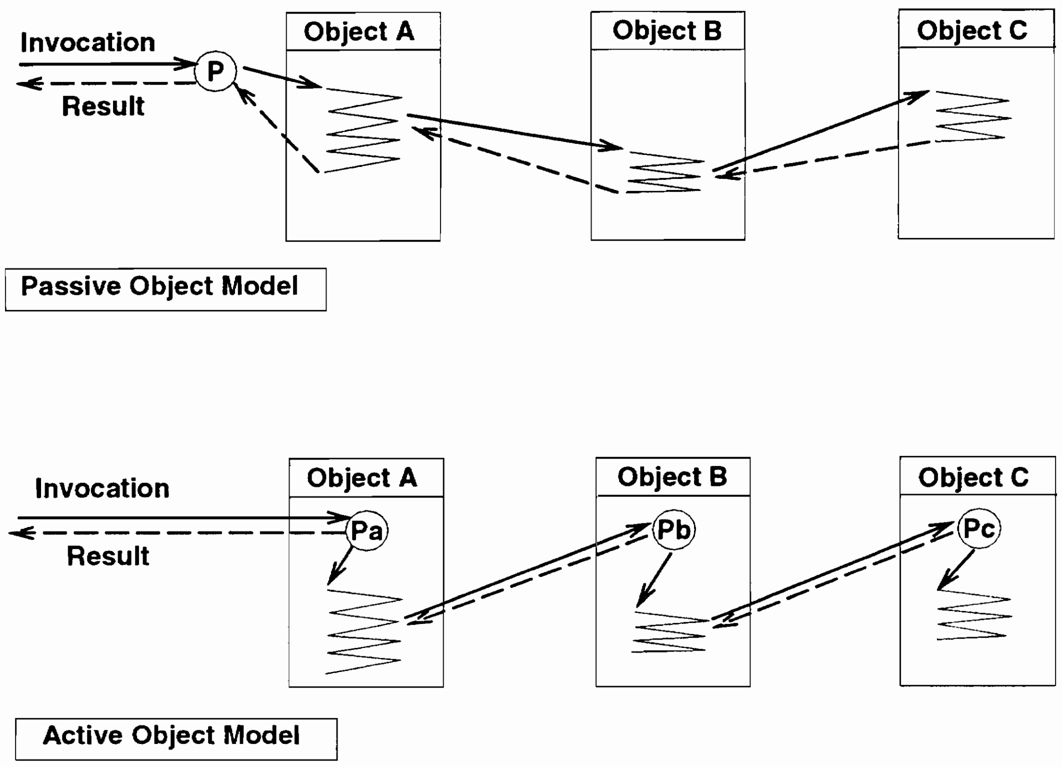


Figure 2.3: Performing an action in the passive and active object model.

The main advantage of using the passive object model is that there is virtually no restriction on the number of processes that can be bound to an object, thus allowing a high-level of object concurrency. However, the cost of mapping a process into and out the address space of multiple objects can be expensive.

The active object model alleviates this problem, but introduces others. In the static model, some processes may always be idle. On the other hand, deadlock can occur. For example, in the case of recursive invocations, an object deadlocks if it does not have enough server processes to handle the requests delivered to it. This problem may be solved in part by the use of the dynamic variation of the model. However, this model has also the inconvenience of adding overhead for the creation and destruction of the processes. Depending on the granularity of the object, this overhead may be non-neglectable.

2.5 Object Interaction Management

When a client wants to invoke the method of an object, the distributed object-based system is responsible for performing the necessary steps to locate the object, execute the method and for returning a result back to the client.

2.5.1 Locating an object

In a distributed system, each object must be assigned a unique identifier. The identifier of an object is constant during its lifetime and, ideally, should never be used again. As a distributed object based system should provide location transparency, the user needs only to know the object identifier and does not have to be aware of the physical location of an object in order to invoke one of its methods.

Whenever an invocation is made, the system should determine which object is concerned and on which workstation it resides. A mechanism should also support the migration of an object to another machine.

Three main approaches to location transparency can be used:

- The name encoding scheme.
- The distributed name server scheme.
- The cache/broadcast scheme.

The name encoding consists in encoding the location of an object in its identifier. This is a very easy to implement and efficient scheme. However, its main drawback is that the object can not migrate, as this would require its name to change. To solve this problem, *forward location pointers* can be used. A forward location pointer is a reference to the new location of an object. Whenever an object migrates, a forward location pointer is created on the initial workstation, to be able to forward the invocation requests. Nonetheless, this mechanism has the drawback of making the system fragile. The failure of one workstation, that contains a forward location pointer, will result in an invocation failure even if the service is really available on another workstation.

The second approach, the distributed name server scheme, consists of having name servers that map the object identifier to its machine location. There may be several name servers, but not necessarily one per workstation. There are two variations of this scheme. First, each server keeps all the information about the objects in the system. This requires a mechanism to avoid inconsistencies between the different servers. Second, one server keeps only partial information, and delegates the requests to the other servers when it is not able to provide the information. The drawback of this scheme, whatever variation is used, is that the creation and destruction of an object, or a method invocation may induce significant communication and delay, either to keep the servers up to date or to find the server able to answer the request.

The third approach consists in keeping one small cache on each workstation, which contains information on a number of most recently referenced remote objects. When a client makes a remote invocation, the cache is examined. If there is an entry for that object, the invocation request is send to the corresponding workstation. However, the object may have migrated making the information contained in the cache invalid, in which case the invocation is returned. If there is no entry in the cache, or if the entry reveals to be invalid, a message is broadcast throughout the network. Every workstation receiving the broadcast examines the table of its local objects and sends a reply message to the originator of the broadcast if it finds the object. The entry in the cache is then updated. This scheme can be very efficient with the use of the cache, and flexible as it allows objects to migrate. It does not require any consistency mechanism. Nevertheless, as all mechanism using caches, the size of the caches must be well tuned, in order to avoid too many broadcasts which greatly affect the throughput of the network.

2.5.2 System-Level Invocation Handling

The way a distributed object based system handles invocations depends entirely on the object model supported. Two schemes used are the message passing scheme and the direct invocation scheme.

Message passing

A system that provides the active object model typically supports the pure message passing scheme to handle object interactions. When a client makes an invocation, the method name and its arguments are packaged into a request message. This message is then sent to a server process or a port associated with the invoked object. A server process in the invoked object accepts the message, unpacks the parameters, and performs the specified operation. When the operation completes, the result is packaged into a reply message, which is then sent back to the client. The main drawback of this scheme is that it involves excessive overhead for communications between objects residing on the same machine.

Direct invocation

The direct invocation scheme is usually used by systems that provide the passive object model. When the two objects that need to communicate are on the same machine, the method is directly invoked as in a centralized system, provided that the target object is in memory. The method invocation is then similar to a usual function call. When the method of a remote object is invoked, a message is sent to the remote workstation, as in the previous scheme. However, the invocation is not handled by one of the processes of the object like in the active object model, but by a worker process, created on behalf of the original process, and killed once the processing is finished and the result has been returned. This scheme is more efficient for local invocations, but is probably less efficient for remote invocations, as the creation and destruction of the worker process involve considerable overhead.

2.6 Advantages of Object-Based Systems for Developing Distributed Applications

The use of an object-based system simplifies the development of a distributed application as objects are autonomous entities that serve well as units for protection, recovery, security, synchronization and mobility.

The prime advantage to using a distributed object-based programming system is that it eases the development of distributed programs. The object abstraction is a natural abstraction used by programmers, and thus, the design and implementation of distributed applications is considerably eased. This is a fundamental characteristic of object-based systems.

2.7 The CORBA architecture

The Object Management Group, an industry consortium, has specified a minimal architecture for distributed object management in an heterogeneous computing environment, called the Common Object Request Broker (CORBA). This section will shortly describe the CORBA specifications. A more detailed description can be found in [NWM93] or [OMG91].

The goal of the CORBA project is to specify the components and the organization of the components needed in order to:

- Locate an object implementation for a request.
- Ensure that an object implementation is in memory, ready to receive a request.
- Transmit the requests and results between the client object and the server object.

The main components of the system as described in the CORBA specifications include:

Objects: Objects are identifiable, encapsulated entities providing one or more services that clients can request. Objects are created and destroyed as a result of executing object requests.

Requests: Requests are the mechanism through which the clients request other objects' services. A request contains the following pieces of information: an operation, a target object, zero or more actual parameters. The target object returns an exception if an abnormal condition occurs during request processing.

Types: Types classify objects according to common characteristics. Types are used to characterize the signature of a method, or its return type. The system should define a set of primitive and compound types found in many high-level programming languages.

Interfaces: Interfaces describe to the rest of the system the operations a client can request of objects.

Operations: Operations are named entities denoting services that can be requested. Every operation has a signature that specifies the arguments required to invoke the operation, the type of result returned, as well as the exceptions that might be returned in case of abnormal processing of the operation.

The CORBA architecture also defines an *Interface Definition Language*, or IDL. The IDL describes the operations and associated attributes of an object interface in a programming language independent form, that the rest of the system can understand. Clients can then use a uniform type of requests, without concern for the way the service requested is implemented.

A client can invoke operations via either the static or the dynamic client interfaces. In the former case, IDL specifications are compiled into client stubs and target-object skeletons. In the latter case, they are stored in an interface repository accessible through the dynamic invocation interface.

In the static invocation interface, there is one stub routine for each particular operation on a particular object that the client may invoke. To invoke an operation on an object, the client calls the appropriate stub routine. This stub routine will transform the language independent request into a language dependent request and send it to the object which method is being invoked.

The drawback of this method is that the interface is static, and that services that are not available at the time of the compilation can not be accessed. However, the client can then use the dynamic invocation interface. The client supplies the request's target object, and the ORB core services return the list of services that can be accessed, using the interface repository. The client can then choose dynamically a service and provide the arguments required by this service to complete his request. The request is then sent to the target object.

It must be noted that all the information contained in the interface repository is in an IDL format. As a consequence, the request, once received by the target object, needs to be translated before performing the execution. This translation is performed by an object adapter. The object repository can also facilitate type-checking at run-time.

An object adapter not only supports the translation of a request from an IDL form to the language in which the object is implemented, the object adapter also provides convenient interfaces to the following functions:

- The dynamic loading in memory of an object implementation when it is not available in memory to satisfy a request.
- The generation and interpretation of object references.
- The registration of object implementations.

The use of the CORBA specifications is intended to facilitate the interoperability of heterogeneous systems. However, this system relies on the use of many adapters. As of now, the number and definition of those adapters is not well-defined. Another problem concerns the dynamic invocation interface. The interface repository is specified as being read only, and thus new services can not be added at run-time. This reduces the capabilities of the dynamic invocation interface.

Chapter 3
THE ACTOR MODEL

3.1 Description of the Model

The actor model was first developed by Carl Hewitt [HBS73] and extended by Gul Agha [Agh86]. Actor languages support objects, data abstraction, and concurrency but not classes or inheritance.

In the actor model, the objects are called "actors". An actor has a mail address (mailbox name), and a *script*, which is the representation of its behavior. The *mail address* is a pointer to a buffer which can store an unbounded number of messages that are received by this actor. The *behavior* corresponds to the actions undertaken by an actor in response to a message. Thus an actor can be modeled as shown in Figure 3.1.

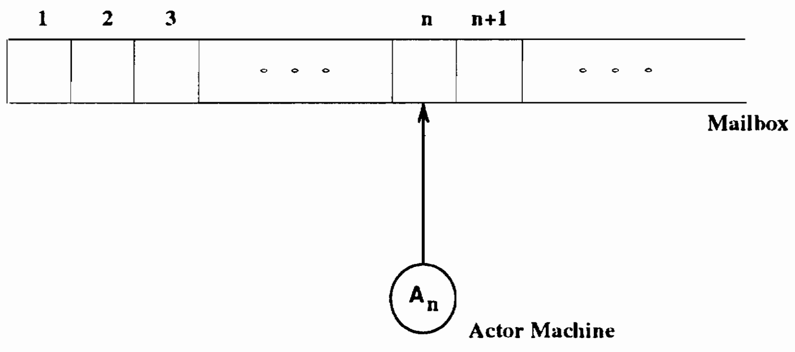


Figure 3.1: Representation of an Actor.

The identity of an actor is determined by its mailbox name, which is firmly decoupled from its state and behavior. Each instance of a behavior can process only one message. If the message queue is empty, when an actor is ready to receive the next message, the actor is blocked until a message arrives in the mail queue. The actions that can be taken by a behavior, as described by Figure 3.2, are the following:

- send a message (send);
- create one or several actors (new);
- specify its transformation (become).

An actor may communicate with a limited number of other actors, for which it knows the mailbox name, by sending messages. Those actors are said to be *acquaintances* of this actor. Communications are asynchronous. This means that an actor which sends a message to another actor may continue to do its own processing between the time its sends the message and the time it receives a reply.

An actor may create new actors. The mailbox name of newly created actors is known to the creating actor, which it can disseminate to other actors by sending messages containing the mailbox name. Note that the creation of an actor is not performed by deriving an instance from a class. Instead, each actor can behave as a *prototype* for the creation of a new actor.

As a behavior can process only one message, a behavior must designate, using the operation become, a *replacement behavior*, which will process the next message.

An actor encapsulates code and data in an active object. Each actor has at least one thread of execution bound to it. Thus, the actor model supports concurrency, which can be observed at three levels. First, at the actor level, there may be several behaviors, each in a different actor, executing actions at the same time. Second, there is also a concurrency at the instruction level. There is no notion of sequentiality for the processing of the instructions of a script. Thus the instructions of a script may execute concurrently. However, this level of concurrency has not been supported yet by the actual implementations of the actor model. An instruction level fine-grain concurrency presupposes machines with a large number of processors and very fast interprocessor communication, which are not available, as of now.

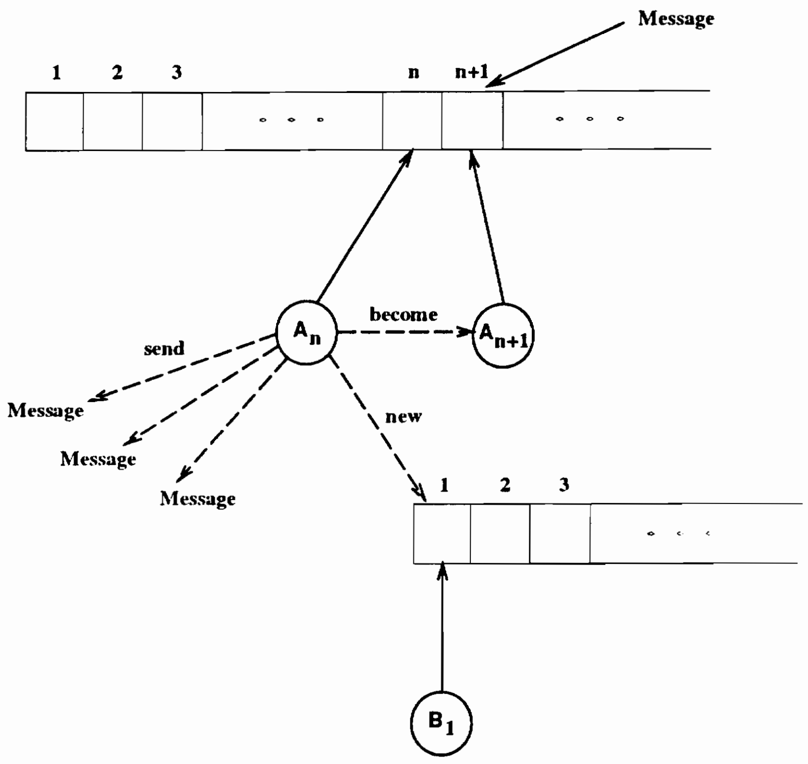


Figure 3.2: Actor operations.

Third, it must be noted that a single actor may have more than one behavior at one time. Depending on when the become operation is performed, the replacement behavior may begin to process the next message concurrently with the creator of the replacement behavior.

The actor model does not support inheritance, which violates, to a certain extent, the principle of encapsulation, but delegation. As said before, delegation offers the same possibilities as inheritance, and has also the advantage of keeping the communication protocol uniform. An actor delegates an action to another actor by sending it a message, and no pointer reference is used as with inheritance.

3.2 An Implementation of the Actor Model : ACT++

One implementation of the actor model on a parallel machine is ACT++ [KL90]. ACT++ is a class library written in C++ that implements the abstraction of the actor model, with slight modifications. The instruction level fine-grain concurrency is not available. Thus, in ACT++, a behavior in execution is a sequential process.

The main components of the ACT++ system are described below.

Objects: Two kinds of objects are available in the ACT++ environment: active and passive objects. As defined in section 2.4, active objects have their own thread of execution and may execute concurrently, whereas passive objects do not. Consequently, the methods of a passive object can only be executed using the thread of the requesting object. In order to avoid inconsistencies in the state of a passive object resulting from concurrent invocations, a passive object should be known to only one active object. However, ACT++ does not provide any checking mechanism to enforce such scoping.

Class: Each object is an instance of a class. An active object is an instance of a class, which inherits, directly or indirectly, from a special class called ACTOR. The other objects are passive objects.

Actor: An actor is an active object. It is created using a predefined operation `New`. The actor created is a pair of a new mail queue and an instance of an actor class, where an actor class is a class that inherits, directly or indirectly, from a special class called ACTOR. An actor script is written using both the actor primitives (new, become, send) and the imperative language constructs of C++.

Message Passing: Communications between active objects is done through asynchronous message passing. There are two types of messages: the request messages and the reply messages. A request message corresponds to a method invocation, while a reply message contains the result of a method invocation. Request messages are addressed to a mailbox of a predefined type `Mbox`, which is the type of the mailbox of an actor.

Reply messages are addressed to a mailbox of the predefined type `Cbox`, which is the type of mailbox used to receive the result of a method invocation. A request message contains the name of the method to be invoked, its parameters, and a Cbox name if a result needs to be returned to the initiator of the call. A reply message just contains the value being returned. Two operations are provided to interrogate a Cbox. When an actor wants to know if a message has been received by a Cbox, it uses the operation `in()`; to get the value contained in the Cbox, it uses the `receive` operation. The receive operation can be used as a synchronization mechanism, as it is a blocking operation, which means that if no reply has been received yet, the actor blocks until a reply arrives.

3.3 A Distributed Actor System

An actor system is a set of actors that interact with each other. In a centralized environment, an actor system can be depicted by a graph, as shown in Figure 3.3. Actors are represented by the nodes, and the possibility for an actor to send a message to another actor by a directed arc. The directed arcs leaving an actor are termed the acquaintances of this actor. An acquaintance is typically a pointer to an actor in a shared memory environment.

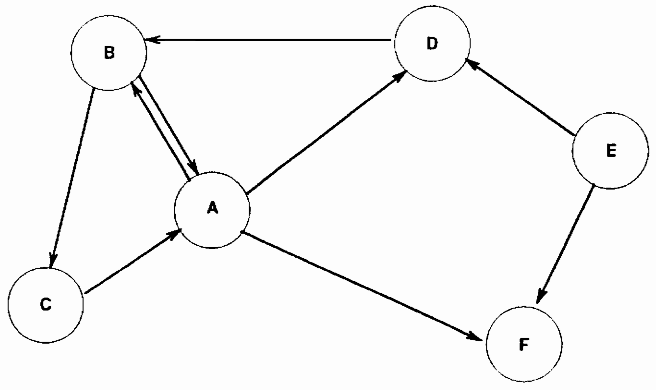


Figure 3.3: A centralized actor system.

However, in the case of a distributed actor system, acquaintances can not be implemented with pointers. An actor identifier scheme must be chosen and support is then needed to generate and interpret actor references. In an heterogeneous environment, a mechanism to encode the messages in an architecture independant format is necessary as different machines may implement data types differently. The sender then encodes a message in this format. The encoded message is sent through the network to the receiver, which decodes the message. Support for dynamic loading and linking of an actor implementation is also necessary, as an actor implementation may be present on several nodes, but used on only one node. The duplication of implementations may be useful to ensure the availability of a service in the case of a workstation failure, or to allow load balancing between different nodes. Finally, actor implementations must be registered to allow the dynamic linking of the object file which is defining an actor. Figure 3.4 represents the previous system, once it has been distributed.

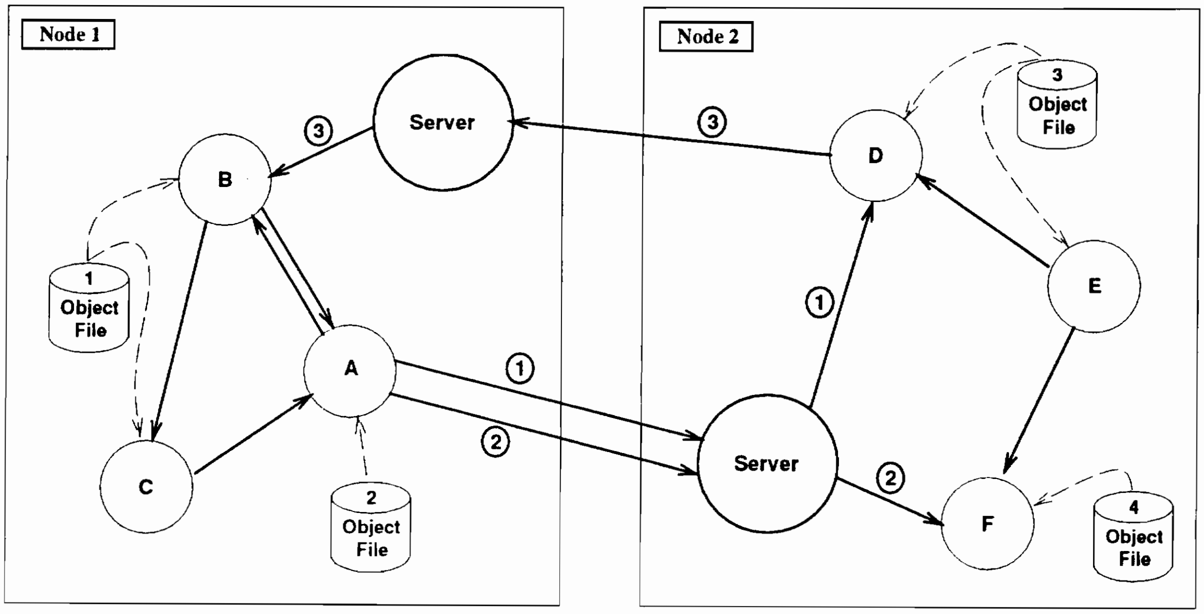


Figure 3.4: A distributed actor system.

Local invocations and actor creations are still processed in the same way as for the centralized system. On each node, a server is present which handles all the messages arriving to a node. The server decodes the message, loads the object implementation in memory if needed, and then invokes the constructor or method. It must be noted that the message encoding and transmission is not represented in this figure. One way to implement the server is presented in the next chapter.

Chapter 4

A DISTRIBUTED ENVIRONMENT FOR THE ACTOR MODEL

This chapter describes the environment that we have created that provides the features described in Section 3.3. However, the message encoding and transmission problems are not addressed, as they are the subject of a related project.

4.1 Overview of the Environment

The environment comprises five main components, as shown in Figure 4.1:

- A run-time loading entity using the facilities of a dynamic linker called "Dld". This component is called `loader`.
- A query service to identify the classes present in an object file, called `class_retriever`. This component is derived from the source-level debugger "GDB".
- A directory service that locates class methods in memory, called `class_directory`.
- A directory service that locates an actor in memory, called `actor_directory`.
- A `server` handling the incoming messages.

The design and implementation of the server is outside the scope of this project. Therefore, only a high-level description of the server's interface with the other components of the system (represented by the dotted line on the Figure 4.1) will be given.

When a message is received by the `server`, it is first decoded to know whether the message is:

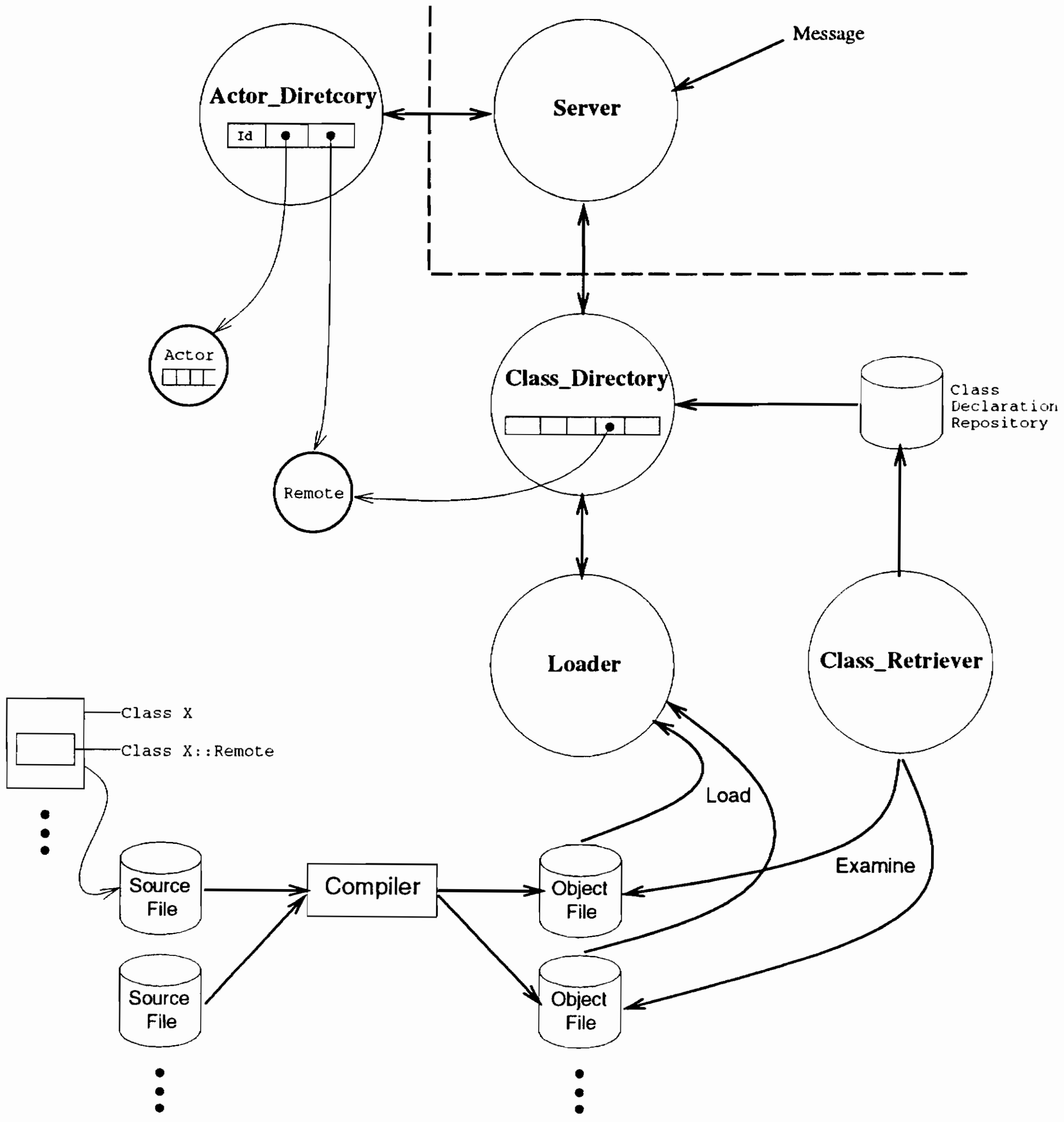


Figure 4.1: Overview of the Environment.

- A request for the creation of an actor.
- A request for the invocation of the method of an actor.
- A request for the destruction of an actor.
- A reply message, which returns the result of a method invocation.

The first three messages are relevant to an actor, while the last one is relevant to a Cbox, which is the mechanism used by ACT++ to return the result of an operation. The actions taken by the server in response to each type of message is briefly described here and presented in more detail in the next section.

In the case of a request for the creation of an actor, the message received has a structure as shown in Figure 4.2. The `server` sends a request to the `class_directory` service. The `class_directory` service first checks if the object module defining the class, from which we want to create an instance, is in memory or not. If it is, the object is then constructed, and the address of the object created is returned. Otherwise, the `class_directory` service sends a request to the `loader` service to load the object module corresponding to the class. An object called `Remote` is also created. There is one such `Remote` object created for each class. This object is always created when the first request to create an object of that class is received. The purpose of the `Remote` object is to handle the decoding of the method invocations arguments (including invocation of constructor methods) for all the objects of that class. Once the object module has been loaded, the `class_directory` proceeds as described above. The server then sends a message to the `actor_directory` to create an entry for the actor just created. The server receives an actor identifier, that it transmits to the initiator of the message using the Reply point contained in the message.

Class_Name	**Constructor Arguments**	**Reply Point**

Figure 4.2: The Constructor Message Format.

The `class_directory` service is initialized using a file called `Class Declaration Repository`. This file is created and updated by the `class_retriever` service, which examines the object files present on one node and determines the classes that are defined in them. The `Class Declaration Repository` contains the information to match class names to object files, that is used by the `class_directory` service. The information contained in `class_directory` is then static once the system is initialized, except if an explicit request to read a new `Class Declaration Repository` is made.

In the case of a method invocation request, the message received has a structure as shown in Figure 4.3. The actor identifier is decoded and sent to the `actor_directory`. The `actor_directory` then retrieves the address in memory corresponding to that identifier and a pointer to the object `Remote`. The server then sends the remaining part of the encoded message (method name and method arguments) to be decoded. Finally, the server sends the decoded message to the actor.

Actor Id	**Method Name**	**Method Arguments**

Figure 4.3: The Invocation Message Format.

In the case of a destructor message, the destructor is called using the same mechanisms as for a regular method invocation. In addition, the server sends a request to the `actor_directory` to remove the entry corresponding to the actor being destructed. The destructor message is similar to a method message, except that a destructor never has any argument as shown in Figure 4.4.

Actor Id

Figure 4.4: The Destructor Message Format.

In the case of a reply message, the `server` delivers the message to the Cbox having the `Cbox_id` contained in the message.

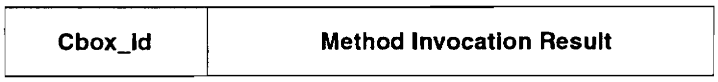


Figure 4.5: The Reply Message Format.

The next section will describe the interface and implementation of each of the components of the system.

4.2 Details of the Current Implementation

4.2.1 The communication interface generated for each behavior

For each behavior, a communication interface is created. This communication interface is in fact implemented by a class definition, called `X::Remote`, where X is a behavior class definition (See Figure 4.1). This is done for two main reasons:

- We needed to transmit the arguments of an invocation or construction and to encode/decode these arguments easily. The use of a pointer to reference the arguments is appropriate in the case of a shared memory architecture, which is the framework in which C++ has operated in the past. However, in the context of a distributed architecture, argument passing can not be based on a pointer mechanism. As a consequence, we need to transmit the arguments themselves, and not a pointer to the arguments.

- We also needed a way to handle overloaded and virtual functions. A typical C++ program makes extensive use of overloaded functions, and it was deemed important to support that feature. Virtual functions is the C++ mechanism that implements the polymorphism property.

To solve these two problems, a "remote interface" is created for an actor as follows. For each behavior definition, named X, a class `X::Remote` is defined, as shown in Figure 4.1. One object of that class is created for all actors executing this behavior definition. This object will then handle all the method invocations concerning these actors. The `Remote` object takes care of argument decoding and, in the case of overloaded and virtual functions, of selecting the right function to be called. This solution has the advantage that no run-time support is needed to handle those two features. The obvious drawback is the generation of the code, which needs to be automatized. The reason for which we have chosen that solution is that we have not found any other inexpensive way to handle overloaded and virtual functions at run-time. To solve the first problem, we have used a tool that is developed in a current project, which implements type information for encoding and decoding.

4.2.2 The server services

The role of the server has already been described in Section 4.1. We will now describe the communications that it performs with the other components in detail.

Description of the interface

Constructor Message

As shown in Figure 4.6, the server receives a message which it determines is a constructor message (Step 0 on the Figure). The `server` first sends the class name to the `class_directory` service (Step 1). In return, it receives a pointer to the object `Remote` corresponding to that class name (Step 2). The server sends the remaining part of the message to the `Remote` object to be decoded(Step 3a). The `Remote` object returns the decoded message which the server uses to construct the behavior of the actor and then the actor itself (Step 3b). It then needs to add an entry in the `actor_directory`, which it does by sending a structure containing a pointer to the actor created and a pointer to the `Remote` object, that corresponds to the actor's behavior (Step 4). The `actor_directory` service returns an actor identifier (Step 5). The server then returns the actor identifier to the initiator of the call, by sending a reply message (Step 6).

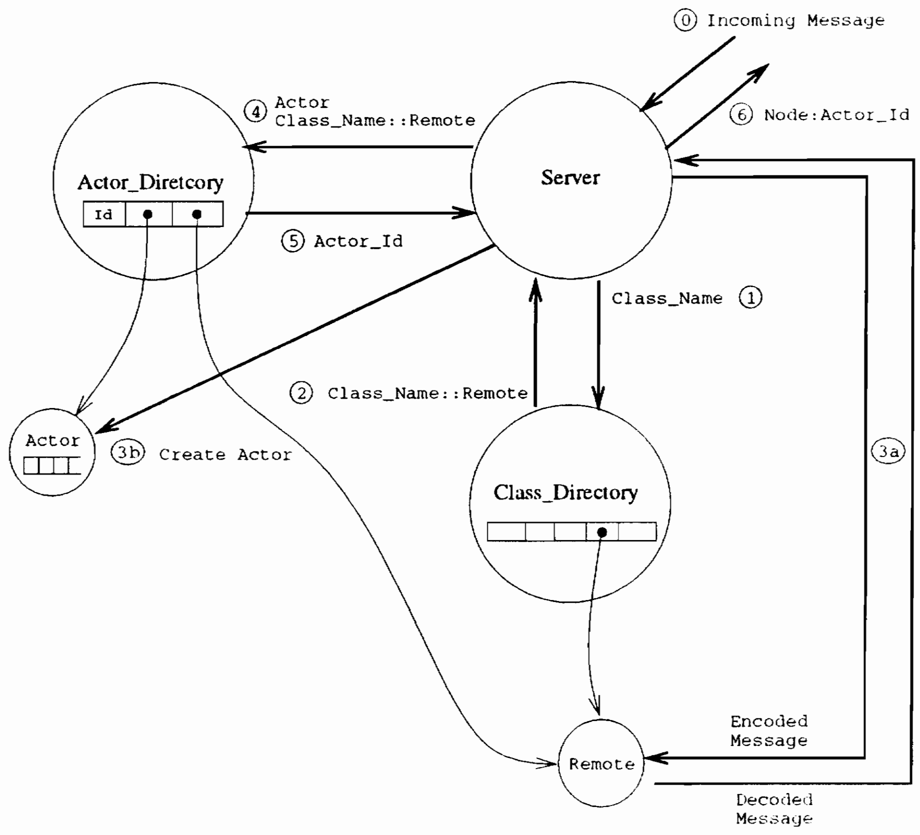


Figure 4.6: The Actor Construction Process.

Method Invocation Message

As shown in Figure 4.7, the server receives a message which, in this case, it determines is an invocation message (Step 0). The server first sends an actor identifier to the **actor_directory** service (Step 1). This returns a structure containing a pointer to the actor previously created and a pointer to the **Remote** object that was used to construct the actor (Step 2). The remaining part of the encoded message is then send to the **Remote** object in order to be decoded (Step 3a). The decoded message is then enqueued in the actor mail queue (Step 3b).

Destructor Message

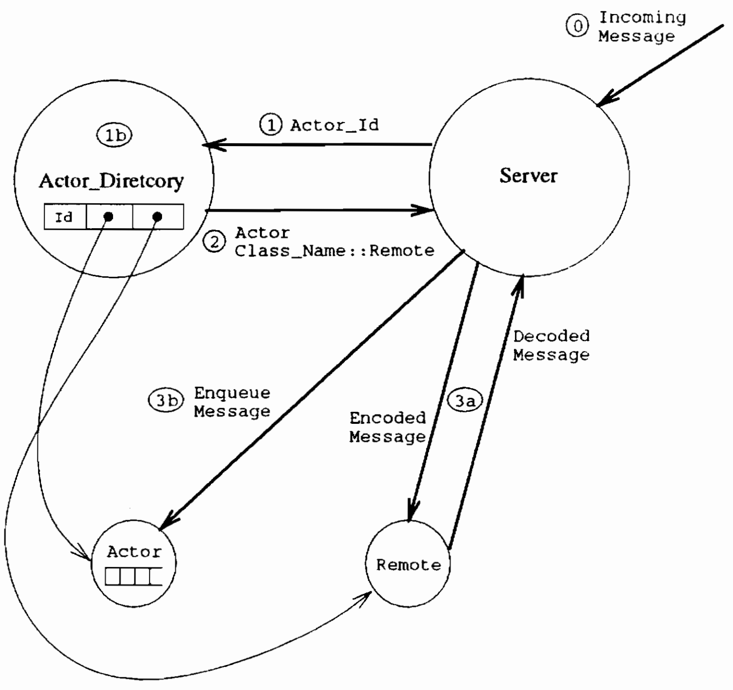


Figure 4.7: The Method Invocation and Actor Destruction Process.

The series of events is similar to the one of a regular method invocation. However, one step is added (Step 1b, Figure 4.7), which is the deletion of the entry of the actor being destructed in the `actor_directory`.

Reply Message

The message is decoded to obtain the `Cbox_id`. The remaining data in the reply message is simply forwarded to the Cbox.

4.2.3 The map class

This template class is used by the `class_directory` and the `actor_directory` services. It defines a template for an associative array, also called map or dictionary. A map keeps pairs of values. The first component is called the key, and the second the value. The main purpose of a map is to retrieve the value corresponding to a key.

Two classes are defined: `Map` and `Mapiter`. The class Map is the class defining the table. The class Mapiter is a friend class of Map, and is a class providing iteration functions to examine the table. Several iterators of the class Mapiter can be instantiated, thus allowing concurrent interrogation of a table.

Part of the implementation of this Map class can be found, along with some comments in [Str91].

4.2.4 The class directory services

The `class_directory` service is the service provided to the server that returns a pointer to an object `Remote` corresponding to a given class. To perform that task, the `class_directory` service uses two other components: `loader` and `class_retriever`. The next section presents the interface of this service in detail.

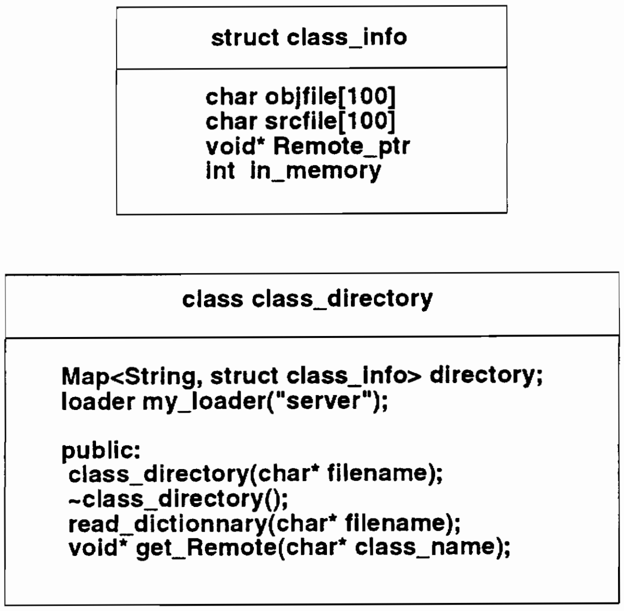
Description of the interface

Figure 4.8 shows the interface of the `class_directory` class.

The class_directory service is implemented using the map class described in Section 4.2.3. The key value is a class name, and the structure retrieved contains the following elements:

- `char *objfile`: The name of the object file defining that class.
- `char *srcfile`: The name of the source file defining that class.
- `void *remote_ptr`: The pointer to the object `Remote` associated with that class. This pointer is set to zero if no requests have been made to create an object of that class.
- `int in_memory`: This field is set to zero if the object file `objfile` is not currently in main memory, and one otherwise.

The methods defined for the class are:



```
struct class_info

char objfile[100]
char srcfile[100]
void* Remote_ptr
int  in_memory
```

```
class class_directory

Map<String, struct class_info> directory;
loader my_loader("server");

public:
 class_directory(char* filename);
 ~class_directory();
 read_dictionnary(char* filename);
 void* get_Remote(char* class_name);
```

Figure 4.8: The class_directory class declaration.

- `void class_directory(char *filename)`
- `void read_dictionary(char* filename)`
- `void* get_Remote(char* class_name)`

The `class_directory` method builds the data structure previously described from a file which contains the elements (class name, object file, source file). This file is created and updated using the `class_retriever` service, described in the next section. The method `read_dictionary` is used to add entries in the table using another file.

The method `get_Remote` takes a class name as an argument and returns a pointer to the object `Remote` corresponding to that class. If such an object does not exist, which means that no object of that class have never been created, the service `loader` is used to get the address of the `Remote` constructor. The object is then created and the table entry updated, before the pointer to the newly created object is returned. The `loader` service is described in Section 4.2.7.

4.2.5 The class retriever services

The `class_retriever` service can create or update the file containing the information needed for `class_directory`. Given an object file name, and its directory path, `class_retriever` will determine the classes that are defined in the object file. The object file must have been compiled with the "-g" option, so that it contains the required symbol table information. Due to a limitation in g++, the current tool does not support nested classes.

The methods defined for this class are:

- `void create_dictionary(char* objfile_filename, char* dict_filename)`
- `void append_dictionary(char* objfile_filename, char* dict_filename)`

The fist method will read the symbol table information of the object file specified and create a new file containing records of the form (class name, object file name, source file name).

The second method is similar, except that the information found in the object file is appended to the existing dictionary file.

The implementation of these methods is based on a debugger called "GDB". A tool was derived from it, which creates a data structure containing the symbol table information. From this data structure, we can then determine what are the classes defined in a particular object file. This tool can be very easily extended to give other information about the symbol table of an object file.

Section 4.2.6 describes the parts of the source-level debugger "GDB" that we used.

4.2.6 GDB, a source-level debugger

GDB is a source-level debugger able to run on a wide variety of platforms running under Unix. It also supports several different executable file formats including the coff and the a.out formats among others. GDB reads the symbol table information from those different file formats and converts it into a common, file format independent data structure. It then uses that data structure to obtain all the information it needs to reply to the queries of a user debugging a program.

The reuse of some parts of GDB was important for two reasons. First, we would not have to rewrite all the non-trivial code for the reading of the symbol table information. Second, as GDB supports several executable file formats, our environment could be ported on different machines with little effort.

In the next section, we will describe the different components involved in the creation of the data structure. We needed to isolate those components in order to build our tool, as a great part of GDB concerned the treatment of breakpoints, display of source code, etc. that were irrelevant to our purpose. Then, we examine the data structure created.

Reading the symbol table information with GDB

GDB uses a library from the Free Software Foundation called "The Binary File Descriptor Library", found in the directory GDB-4.X/bfd under the name libbfd.a. BFD is in fact the part of GDB that allows the support of different file formats. BFD offers a set of routines to operate on object files whatever the object file format is. Simply, BFD can be split in two parts, the front end and the many back-ends. The front end of BFD is the set of services offered to the user. It manages memory, and various canonical data structures. The front-end is also responsible for which back-end should be used. A back-end is just the implementation of the services for a particular object file format. It consists of a set of functions, and of some information proper to that object file format, used for greater efficiency. BFD provides support for GDB in several ways:

object file format identification: BFD will identify a variety of file types, including a.out, coff, and several variants thereof.

access to sections of files: BFD parses the file headers to determine the names, virtual addresses, sizes, and file locations of all the various named sections in files (such as the text section or the data section). GDB simply calls BFD to read or write section X at byte offset Y for length Z.

specialized core file support: BFD provides routines to determine the failing command name stored in a core file, the signal which the program failed, and whether a core file matches (i.e. could be a core dump of) a particular executable file.

locating the symbol information: GDB uses BFD to determine where to find the symbol information in an executable file. GDB handles itself the reading of symbols, since BFD does not understand debugging symbols.

GDB opens symbol files with symfile_bfd_open (GDB-4.X/gdb/symfile.c) using the BFD library. BFD identifies the type of the format of the object file by examining its header. This is achieved by trying several file formats using the routine bfd_check_format (GDB-4.X/bfd/targets.c), which returns true when the guess was right.

For each format identified, the corresponding routine _initialize_xxxread (GDB-4.X/gdb/xxxread.c), where xxx is the file format, is called. This routine then invokes add_symtab_fns (GDB-4.X/gdb/symfile.c), which creates a structure sym_fns (see Figure 4.9) and links it to the global sym_fns list. This structure registers information about each format GDB is prepared to handle.

This structure contains the name (or name prefix) of the symbol format, the length of the name, pointers to five functions, and a link to the next sym_fns structure. Those functions are called at various times to process symbol files whose identification matches the specified name.

The functions, contained in GDB-4.X/gdb/xxxread.c, supplied by each structure are:

xxx_symfile_init(struct objfile*) This function is called by symbol_file_add (GDB-4.X /gdb/symfile.c) when we are about to read a new symbol file. It performs initializations and cleanups, in order to read a new symbol file. Note that the symbol file which we are preparing to read might be a new "main" symbol file, or might be a secondary symbol file whose symbols are being added to the existing symbol table.

Struct sym_fns
symfile.h

```
char                       *sym_name
int                        sym-namelen
void                       (*sym_new_init)
                           PARAMS((struct objfile*))
void                       (*sym_init)
                           PARAMS((struct objfile*))
void                       (*sym_read)
                           PARAMS((struct objfile*, struct section_offsets*, int))
void                       (*sym_finish)
                           PARAMS((struct objfile*))
struct section_offsets*    (*sym_offsets)
                           PARAMS((struct objfile*, CORE_ADDR))
struct sym_fns             *next
```

Figure 4.9: The GDB data structure : sym_fns structure

xxx_new_init(struct objfile*) This function is called by symbol_file_add (GDB-4.X/gdb/symfile.c) when we want to discard the existing symbols in order to read a new symbol file.

xxx_symfile_read(struct objfile*, struct section_offsets*, int) This function is called by symbol_file_add (GDB-4.X/gdb/symfile.c) to actually read the symbols from a symbol file into a set of partial_symtabs structures. GDB reads the symbol table information in two passes. The function xxx_symfile_read only does the minimum work necessary for letting the user "name" things symbolically and does not read the entire information. It creates partial symbol tables which are mutated into full symbol tables when more information is needed. This is done by the function xxx_psymtab_to_symtab (struct partial_symtab*), which goes back and reads the symbols completely. This function is called by psymtab_to_symtab (GDB-4.X/gdb/symfile.c) or the PSYMTAB_TO_SYMTAB macro. The reading of the symbol table information in two steps is done by GDB for efficiency reasons. As a matter of fact, it is useless and inefficient (in time and space) to read the full information about some symbols if the user of the debugger doesn't ask any information about those symbols. This is often true for the code of a library for example.

xxx_symfile_finish(struct objfile*) This function is called to perform any local cleanups required when we want to discard all the symbol table information for a particular object file, freeing up all the memory held for it, and unlinking the object file from the global list of known object files.

struct section_offsets* xxx_symfile_offsets (struct objfile*, CORE_ADDR) This function is called when a new module is dynamically linked. Given an address of where the new module has been loaded, GDB creates a structure called section_offsets in order to increase the speed of reading the symbols. The structure is a set of amounts by which the sections of this object file were relocated when it was loaded in memory. All symbols that refer to memory locations need to be offset by those amounts.

This short description of the mechanisms involved in the reading of the symbol table information has introduced several structures that have not been defined completely, even if the names used were usually very explicit of what they were representing. A detailed description of the data structure created by GDB can be found in Appendix A.

4.2.7 The loader services

Description of the interface

Figure 4.10 shows the interface of the loader class.

In order to be able to link dynamically an object class for which a method invocation is requested, we first need to create an object of the class loader. The constructor of that class takes a filename as an argument. That filename must be the name of the executable file with which the new module will be dynamically linked. This is the initialization phase. If the initialization fails, an error message is written to an error log.

Then, in order to link dynamically the new module, the method `link` of the loader object must be invoked. This function takes the name of an object file as an argument. If the new module is linked in successfully, the function returns one. Otherwise, it returns zero and writes an error message on the error log.

```
class loader

private:

ofstream errorlog;
void open_errorlog ();
void write_errorlog (char* error_message, char* args = "");
void write_errorlog (char* error message, int args);

public:

loader (char* filename = "");
~loader();
link (char* filename = "");
void* find_function (char* function_name = "");
unlink_by_symbol (char* symbol_name = "", int hard = 0);
unlink_by_file (char* filename = "", int hard = 0);
```

Figure 4.10: The loader class declaration.

The next step is then to get the address of the constructor for the class `class_name ::Remote`, where `class_name` is the name of the class for which we want to create an object. This is done by invoking the method `find_function`. This method takes one argument, the mangled name of the method for which we want the address. If the operation is a success, the function returns the address of the method and zero otherwise. This operation may fail for two reasons:

- The method can not be found in the function space. This may occur if we have forgotten to load the module containing the method using the `link` method, or if we have misspelled the name of the method. In this case, an error message indicating that the function has not been found is written on the error log.

- The method may not be executable, which means that some references contained in the method are not resolved. In this case, an appropriate message is written to the error log, with the number of symbols undefined and their names.

When a class definition is not needed any longer because there are no objects of that class remaining, we may want to unlink the corresponding object module. This can be done using the method `unlink_by_symbol` or `unlink_by_file`. Those two methods perform the same operation, but take different arguments. The method `unlink_by_symbol` unlinks the object module that defines a given symbol, whereas `unlink_by_file` unlinks the object module which has a given name. Both methods have a second argument, a flag called hard, which selects between two unlink modes described in section 4.2.8.

All those methods are implemented using a dynamic linker called Dld, described in the next section.

4.2.8 Dld, a dynamic linker

Dld is a library package of C functions that performs dynamic link editing that has been developed at the University of California at Davis by W. Wilson Ho and Ronald A. Olsson. Programs that use Dld can add compiled object code to or remove such code from a process anytime during its execution. Loading modules, searching libraries, resolving external references, and allocating storage for global and static data structures are all performed at run-time. A more detailed description of Dld can be found in [Ho91a].

The use of the Dld library package is straightforward, and can be summed up by the following steps:

1. The header file dld.h must be included in any program that needs to use the functions of Dld.

2. Dld needs to be initialized first, in order to initialize some internal data structures, and to load in memory the symbols definition of the executing process. This initialization must obviously take place before any other function of Dld is invoked.

3. The desired relocatable object file or library is then loaded into memory. Dld tries to resolve as many undefined external references as possible.

4. We can then get the entry point of a function or the address of a symbol given its

name. This information can be used to invoke a function defined in a module just linked in.

5. Finally, we can unlink a module.

The unlink operation

As said before, a module can be unlinked using one of the two unlink modes provided by Dld. Those two modes are:

1. Soft unlink (hard = 0). In this mode, the module is marked as removable, but it is actually removed from memory only if it is not referenced by any other module.

2. Hard unlink (hard is non-zero). In this mode, the module is removed from memory unconditionaly.

This is illustrated by the Figure 4.11, as found in [HO91b].

Three modules have been linked into memory (Figure 4.11 (a)). The modules A and B have both a reference to some symbol defined in module C. Then, we want to unlink module A. When we do so, a process of garbage collection starts to remove all the modules that were dependent on the module that we are removing. However, here we may want to keep module C, even if it is referenced by module A, as it is also referenced by module B. The Figures 4.11 (b) and 4.11 (c) show the effect of the two unlink modes.

The C++ mangling scheme

As said in Section 4.2.7, the method `find_function` requires the mangled name of the function for which we want to get the address. It was then necessary to understand how the function names are mangled. This encoding is compiler dependent. A description of the mangling scheme of the AT&T C++ compiler can be found in [ES90] section 7.2.1c, which is the reference manual of C++ accompanied of some comments on the design choices and implementation of this compiler.

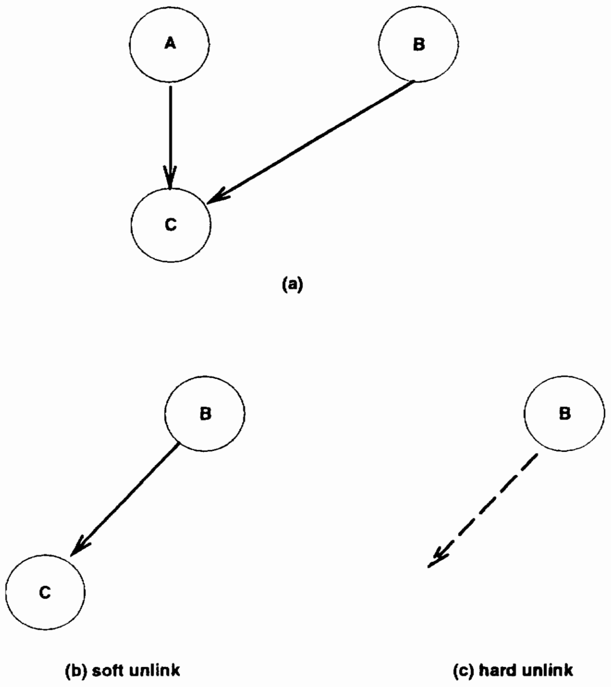


Figure 4.11: Dld: illustration of the two different unlink modes.

However, as we were using the GNU g++ compiler, the mangling scheme is a little bit different than the one described in that book. Here is a short description of the mangling scheme of the g++ compiler. This description is not complete, but is intended rather to give a cursory introduction of how the mangling is done.

The function names were mangled originally to implement the feature of overloaded functions. Obviously, those function names needed to be mangled to make them unique, as only distinct symbols can coexist in the same object file. If the type of the arguments, or the number of arguments differ, then the compiler can mangle the function name to make it unique, and therefore be able to invoke the right instance of the overloaded function for each call.

The name mangling consists in including in the name the signature of the function: the types of its arguments, along with the class name to which it belongs if any. It must be noted that the return type of the function is not present in the mangled name. As a consequence, it is forbidden to have overloaded functions that would differ only in their return type. The name of the function is separated from its signature by a double underscore. Thus, a function name should not contain any such sequence. Each type of the arguments is represented by a letter, as shown in Table 4.1.

Table 4.1: The type encoding.

type	encoding
void	v
char	c
short	s
int	i
long	l
float	f
double	d
...	...

A global function `f(int, double)` is encoded as follows `_f__Fid`. The letter F before the signature of the function indicates that we are dealing with a global function, and not a method from a class. A function taking no argument has a signature v(void). For example, a global function `f()` would become `_f__Fv`.

A method `f(int, double)` belonging to a class called base would be encoded as follows `_f__4baseid`. The number 4 indicates the length of the class name that follows it. This number may have several digits, which is not really a problem, as there is no class name beginning with a number.

The class name is encoded differently for a class declared inside another one. The qualified name of the class must be used in order to avoid any ambiguity. For example, if there exists a class `X` defined in a class `YY` (qualified name `YY::X`), and a class `X` defined in a class `ZZ` (qualified name `ZZ::X`), we need to be able to distinguish those two `X` classes. A qualified name is encoded by a Q (for "qualified"), followed by a single digit, indicating the number of names, followed by the names. A class `YY::X` would then be encoded as follows

Q22YY1X, and a similar function f belonging to that class _f__Q22YY1Xid.

So far, the class name was appearing as the name of the class in which the method is defined. However, a class can be an argument too. A global function f(base), where base is a class name, is encoded as follows _f__FG4base. The letter G indicates that the following argument is a class name. However, that letter is omitted if the method being described belongs to that class.

Type modifiers can also be included in the mangled name.

The arguments that are pointer to, or reference to a type are respectively preceded by P and R. So, a method f(int*, char&, XX&) belonging to a class called base, where XX is a class name would be encoded as follows _f__4basePiRcR2XX.

For arguments that are pointers to function types, the function return type, as well as the argument types are encoded. The encoded return type appears after the argument types, preceded by a single underscore. For example, the following global function f(int (*)(char*)) would be mangled to give _f__FPFPc_i.

To shorten encodings, repeated types in an argument list need not to be repeated in full in the name encoding; instead a reference to where the first occurrence of the type in the argument list can be used. For example, a global function f(complex, complex) would be encoded as _f__FG7complexT0. The T0 means that the argument of rank zero (1st argument) is repeated. Similarly, f(complex, complex, complex) would be encoded as _f__FG7complexN20, where N20 means that the next two arguments are the same as the one of rank zero.

Operator functions are mangled differently. For example, the operator function operator +(int), defined in a class called base gives ___pl__4basei. In this name, pl (for plus) means the operator "+". For each operator, there is a corresponding generic name. The three underscores preceding the name indicates that the function being defined is an operator, and avoid any ambiguity with a user function having a name corresponding to an operator generic name.

Constructors and destructors are also operators. A constructor is defined by three underscores followed by the encoded class name and signature. A constructor `base(int)` for a class base would be encoded as `__4basei`. A destructor is encoded by two underscores, followed by a dollar sign followed by another underscore and the encoded class name. For example, a destructor for class base would be encoded as `__$_4base`. Note that a destructor has no signature, as destructors cannot be specified with parameters.

4.2.9 The class directory services

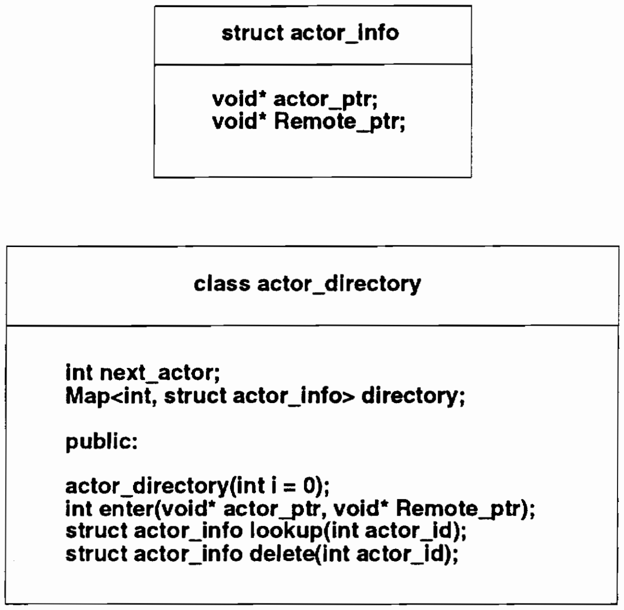
On each machine, a local database keeps track of all the actors that have been created locally. There is no need for a global database, as the machine location of an object is encoded in its identifier. The name encoding scheme (see Section 2.5.1 was used for the creation of an object identifier , as this scheme is simple and efficient. Moreover, the support for object migration is not available for our environment, and consequently more elaborate schemes like the distributed name server scheme or the cache/broadcast scheme (see Section 2.5.1) were not needed.

However, a local database is still needed to map an actor identifier to its address in memory. The actor identifier contained in the database is not the identifier returned to a client. The identifier is first parsed to know on which machine the actor resides, and then the remaining part is used to map it to an address in memory. The identifier then consists of a node name and an integer. The `actor_directory` service assigns an integer to each actor created on the system. The `actor_directory` just keeps a counter which represents the identifier of the next actor to be created. When it receives a request to add an entry in the database, the value of the counter is bound to this actor, and returned to the server. The counter is then incremented.

Description of the interface

Figure 4.12 shows the interface of the `actor_directory` class.

The methods defined for that class are:



```
struct actor_info

void* actor_ptr;
void* Remote_ptr;
```

```
class actor_directory

int next_actor;
Map<int, struct actor_info> directory;

public:

actor_directory(int i = 0);
int enter(void* actor_ptr, void* Remote_ptr);
struct actor_info lookup(int actor_id);
struct actor_info delete(int actor_id);
```

Figure 4.12: The actor_directory class declaration.

- `int enter(void* actor_ptr, void* Remote_ptr)`
- `struct actor_info lookup(int actor_id)`

The first method is used to create an entry in the actor directory. A new identifier (actually an integer) is created for that actor and returned to the caller.

The second method returns the structure actor_info corresponding to the actor identifier given. The structure actor_info contains a pointer to the actor in memory, as well as a pointer to the Remote object that corresponds to the class from which the behavior of this actor has been created.

Implementation

The `actor_directory` service is implemented using the map class described in Section 4.2.3. The key value is an actor identifier on this node, and the structure `actor_info` retrieved contains the following elements:

- `char *actor_ptr`: The address of the actor object.
- `void *remote_ptr`: The pointer to the object `Remote` associated with the actor. Note that many actors may point to the same `Remote` object.

4.2.10 Server/Clients communication

As stated before, the message creation on the client side, and the transfer of the message is part of another related project, not completed yet.

As a consequence, the communication between the server and the clients is actually simulated by the use of a single pipe. The message is created by the client side and sent through the pipe. This does not change anything concerning this project, except that the server is reading the incoming messages from the standard input instead of a socket.

Chapter 5

RESULTS AND FUTURE RESEARCH

The system described in this project performs the following actions:

- The dynamic loading in memory of a required object implementation when it is not available in memory.
- The generation and interpretation of object references.
- The registration of object implementations.

The system supports the feature of object autonomy (see Section 2.3). An actor can invoke a service of another actor only if this other actor belongs to its acquaintance list. Therefore, invocations of the methods of an actor is restricted to the actors having this particular actor as an acquaintance.

The environment has been created on a Sun Sparc workstation, but it should be portable very easily to other platforms. As the tool derived from GDB supports multiple object file formats, there should not be any problem to create a `class_retriever` service on another platform. The `class_directory` service may need to be slightly modified. To get the address of the `Remote` object, the `class_diretory` needs to use the mangled name of the function. As the mangling scheme is compiler dependent, the argument passed from the `class_directory` to the `loader` may need to be modified. The other components should be completely portable.

As of now, the system developed is rather fragile and support for error handling should be added. For example, no data integrity is supported in the current system, leaving an object in an invalid state if an invocation does not complete successfully. If this system supports persistent objects in the future, a mechanism should allow the recoverability of such objects in case of a workstation failure.

One enhancement of this environment could be the automatic unlinking of the object modules. An object module is loaded into memory when a client requests the creation of the first instance of a class defined in this object module. However, there is no mechanism to unlink an object module that would not be needed any more, as all the instances of the classes defined in this module have been destructed. This could be implemented with the help of the object `class_name::Remote`. This object handles all the method invocations for all the objects of a particular class, including the construction and the destruction of those objects. It would then be easy to keep in this `Remote` object a reference counter keeping track of the number of objects currently referencing an object class. When the reference counter reaches zero, a soft unlink request would be initiated. A soft unlink would be used instead of an hard unlink (see Section 4.2.8), because an object module may contain the definition of several classes. The object module should then be unlinked only if the no objects from another class reference this object file. This is exactly the behavior adopted by a soft unlink operation. Some other mechanism could also be used to disable the unlinking process. Some modules may be needed frequently and still have periods where they are not referenced. In this case, we would certainly want to avoid unnecessary unlink and link operations.

One may interject that the environment does not provide complete location transparency, as the client needs to know where the object implementation resides in order to invoke the constructor of this object. In fact, one has to indicate the machine on which the object is to be created. This may happen for two reasons:

- The replication of a service will ensure availability of the service in the case of a workstation failure.

- In the absence of an object migration capability, it may be desirable to create an object on the machine that has the lowest load.

However, if the system supports the properties of availability and load balancing in the future, the client should not have to say where the object is to be created. Nevertheless, once an object is created, no information about its location is needed in order to make an invocation, and for an actor which has not created this object, the location transparency feature is complete.

The support for availability and load balancing would necessitate the creation of a global database for the class directory. This is the only way the system could know where the different copies are located, information necessary to locate an alternative implementation for a service in case of a workstation failure, or to balance the load.

The tools derived from GDB and Dld could be used, with slight modifications, to support the dynamic invocation interface as described in Section 2.7. The creation of the interface repository could be done using the information contained in the data structure created by GDB. For each class, a list of the methods along with the argument list and the returned type can be created. An interface should be created in order to allow the client to choose one method and to supply the arguments required for that particular function. The method could then be invoked using the facilities of the loader component.

In fact the system developed in this project implements many of the facilities described in the CORBA architecture, although this project is involving only one language. The objects, requests and types are similar to the concepts defined in the CORBA specifications. However, there is no support for multiple languages in the current system.

The work done here could also be useful in a centralized object-based environment. A software engineering environment could be developed with the tools derived from GDB and Dld. Using an interface repository, a user could load dynamically a class, create an object of that class and invoke methods for testing purposes. The dynamic linking and unlinking offers means to change quickly the implementation of a class, without having to recompile the entire application. When a class implementation needs to be changed, in order to improve the execution time, or simply to fix a problem, the definition of the class can be unlinked. As a result, the services provided by this class are unavailable for a short period of time, until the new definition is linked in. However, the changes must affect only the implementation of the methods, and not the attributes of a class. Similarly, new functionalities could be added by the definition of new methods, but the original methods must be kept. These restrictions are important in order to ensure that the objects created are still usable after the new implementation has been loaded. If the new implementation is not reliable enough, the old implementation could be restored using the same process in a minimal time.

REFERENCES

[Agh86] Gul A. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems.* The MIT Press, Cambridge, Massachusetts, 1986.

[BGM89] Gordon S. Blair, John J. Gallagher, and Javad Mailk. Genericity vs inheritance vs delegation vs conformance vs... *Journal of Object-Oriented Programming*, pages 11–17, September/October 1989.

[CC91] Roger S. Chin and Samuel T. Chanson. Distributed object-based programming systems. *ACM Computing Surveys*, 23(1):91–124, March 1991.

[ES90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wensley, Menlo Park, California, 1990. ANSI base document.

[HBS73] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd IJCAI*, pages 235–245, 1973.

[Ho91a] W. Wilson Ho. *Dld.* Davis, California 95616, 3.2.3 edition, 1991. Available with the distribution of Dld that can be obtained via anonymous FTP at prep.ai.mit.edu.

[HO91b] W. Wilson Ho and Ronald A. Olsson. An approach to genuine dynamic linking. *Software - Practice and Experience*, 21(4):375–390, April 1991.

[JZ91] Ralph E. Johnson and Jonathan M. Zweig. Delegation in c++. *Journal of Object-Oriented Programming*, pages 31–34, November/December 1991.

[KL89] Dennis Kafura and Keung Hae Lee. Inheritance in actor based concurrent object-oriented languages. *Computer Journal*, 32(4), August 1989.

[KL90] Dennis Kafura and Keung Hae Lee. Act++: Building a concurrent c++ with actors. *Journal of Object-Oriented Programming*, pages 25–37, May/June 1990.

[NWM93] John R. Nicol, C. Thomas Wilkes, and Frank A. Manola. Object orientation in heterogeneous distributed computing systems. *Journal of Object-Oriented Programming*, pages 57–67, June 1993.

[OMG91] OMG, Framingham, Massachusetts. *The Common Oject Request Broker: Architecture and Specification*, August 1991. Document Number 91.8.1.

[Sau89] John H. Saunders. A survey of object-oriented programming languages. *Journal of Object-Oriented Programming*, pages 5–11, March/April 1989.

[Ste87] Lynn Andrea Stein. Delegation is inheritance. In *Proceedings of the conference on Object-Oriented Programming Systems Languages and Applications*, pages 138–146, New York, December 1987. ACM.

[Str91] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wensley, Menlo Park, California, second edition, 1991.

[Weg87] Peter Wegner. Dimensions of object-based language design. In *Proceedings of the conference on Object-Oriented Programming Systems Languages and Applications*, pages 168–182, New York, December 1987. ACM.

Appendix A

DESCRIPTION OF THE "GDB" DATA STRUCTURE

Figure A.1 gives an overview of the data structure created by GDB.

We will give a detail description of each element and its fields, but we will first describe each component at a high-level. The structure `objfile` is the central component of that data structure. It keeps track of each input file from which GDB reads symbols. One structure is allocated for each such file we access, e.g. the executable file and any shared library object files. For each `objfile` structure, two levels of information are available, as depicted before, represented by the structures symtab and partial_symtab. For each source file, there is a corresponding partial_symtab. This contains the information on where in the executable the debugging symbols for a specific file are, and a list of names of global symbols which are located in this file. When more information is needed, the structure symtab is created, but the partial_symtab remains.

A.1 The Objfile Structure

The description of an `objfile` structure is shown in Figure A.2.

The fields of in the `objfile` structure are as follows:

next All the object files are chained together by their next field. A global variable object_files points to the first link in this chain.

name This is the object file's name.

flags Some flag bits for this objfile.

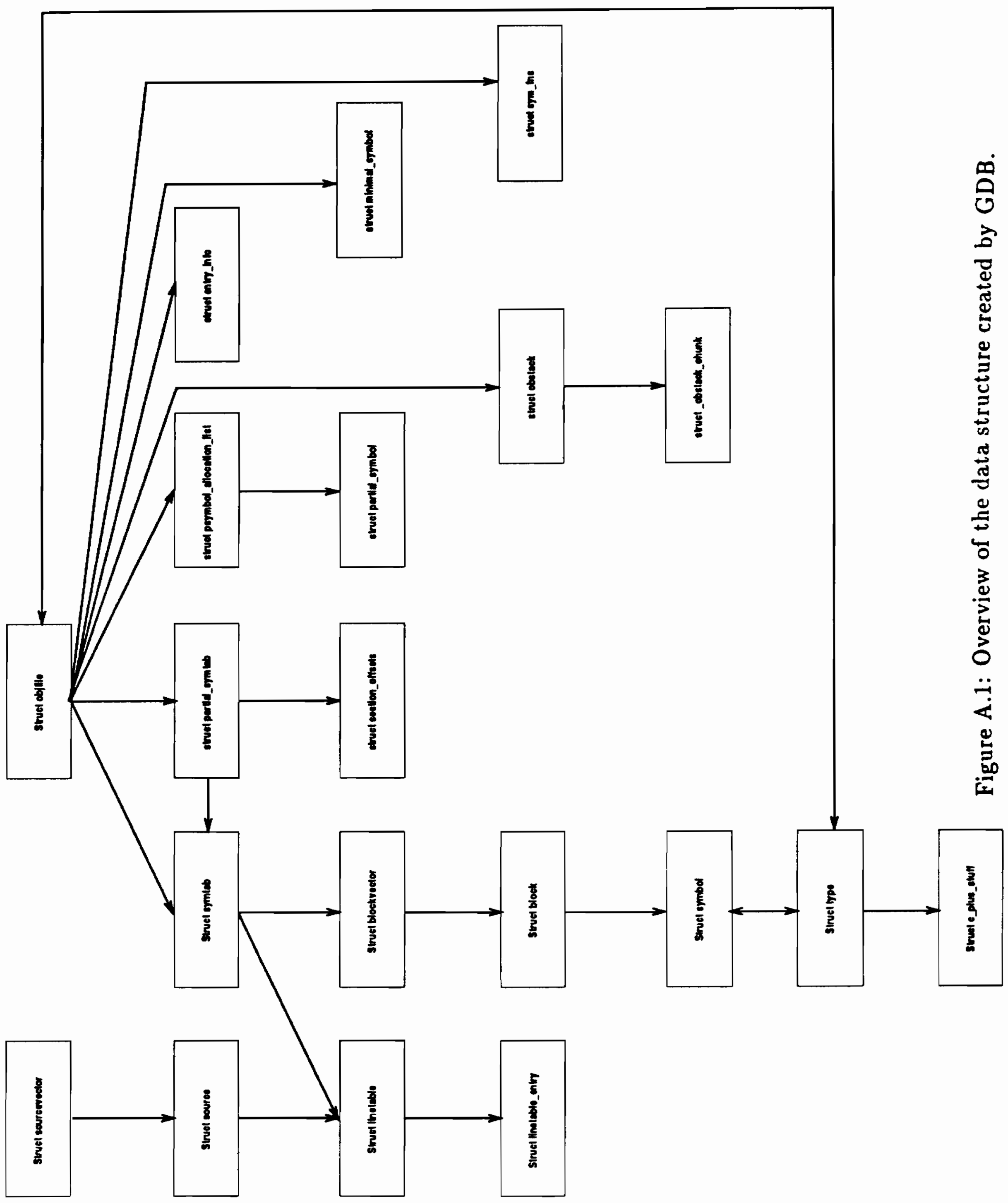


Figure A.1: Overview of the data structure created by GDB.

Struct objfile
objfiles.h

```
struct objfile                   *next
char                             *name
unsigned short                   flags
struct symtab                    *symtabs
struct partial_symtab            *psymtabs
struct partial_symtab            *free_psymtabs
bfd                              *obfd
long                             mtime
struct obstack                   psymbol_obstack
struct obstack                   symbol_obstack
struct obstack                   type-obstack
struct psymbol_allocation_list   global_psymbols
struct psymbol_allocation_list   static_psymbols
struct minimal_symbol            *msymbols
int                              minimal_symbol_count
struct type                      **fundamental_types
PTR                              md
int                              mmfd
struct sym_fns                   *sf
struct entry_info                ei
PTR                              sym_private
```

Figure A.2: The GDB data structure: objfile structure

symtabs Each `objfile` points to a linked list of symtabs derived from this file. There is one symtab structure for each compilation unit (or source file). Each element in the symtab list contains a backpointer to this objfile.

psymtabs This is comparable to the previous field but applies to partial_symtabs.

free_psymtabs This points to a list of freed partial_symtabs available for re-use.

obfd This is a handle to the representation of this object_file as created by the Binary File Descriptor library. We will not describe the data structure bfd that can be found in bfd.

mtime The modification timestamp of the object file, as of the last time we read its symbols.

psymbol_obstack, symbol_obstack, type_obstack These elements are pointers to chunks of memory, called obstacks, allocated in order to store the objects read from an object file. They point respectively to the obstack for the partial_symbols, the symbols and the types.

global_psymbols, static_psymbols These are vectors of all the partial symbols read in from an object file. The actual data is stored in the psymbol_obstack.

msymbols, minimal_symbol_count Each file contains a pointer to an array of minimal_symbols for all the global symbols that are defined within the file. The array is terminated by a "null" symbol, one that has a NULL pointer for the name and a zero value for the address. This is useful when a routine is passed a pointer to somewhere in the middle of the array as an argument. The variable minimal_symbol_count is the number of elements contained in this array, including the "null" symbol. The array itself, as well as all the data that it points to, should be allocated on the symbol_obstack for this file.

fundamental_types Some object file formats don't specify fundamental types. GDB can create such types. It then maintains a vector of pointers to these internally created fundamental types on a per object file basis.

md This is the mmalloc() malloc-descriptor for this object file, if we are using the memory mapped malloc() package to manage storage for this object file's data. Otherwise, this field is NULL.

mmfd This is the file descriptor that was used to obtain the mmalloc descriptor for this object file.

sf This is a pointer to a structure that keeps track of the functions that manipulate object files of the same type as this object file. Note that this structure is in statically allocated memory, and is shared by all the object files that use the object module reader of this type.

ei This contains a per object file information about the user's main() function.

sym_private This is a pointer to an allocated chunk of memory, containing information specific to the particular format of this object file. This information may be used by the functions defined by the sf field (sym_new_init, sym_init, sym_read and sym_finish).

A.2 The Partial_Symtab Structure

The definition of partial_symtab is shown in Figure A.3.

Struct partial_symtab

symtab.h

```
struct partial_symtab   *next
char                    *filename
struct objfile          *objfile
struct section_offsets  *section_offsets
CORE_ADDR               textlow
CORE_ADDR               texthigh
struct partial_symtab   **dependencies
int                     number_of_dependencies
int                     global_offset
int                     n_global_syms
int                     static_offset
int                     n_static_syms
struct symtab           *symtab
void (*read_symtab) PARAMS((struct partial_symtab*))
char                    *read_symtab_private
unsigned char           readin
```

Figure A.3: The GDB data structure: partial_symtab structure

The fields in the partial_symtab structure are as follows:

next All the partial_symtabs are chained through their next pointers.

filename This is the name of the source file this partial_symtab is defining.

objfile This is a backpointer to the object file for which symbols will be read.

section_offsets This is the set of relocation offsets to apply to each section.

textlow, texthigh Those two numbers represent the range of text addresses covered by this file.

dependencies, number_of_dependencies This is an array of pointers to all of the partial_symtab's on which this one depends. This array can only be set to previous or the current symtab.

globals_offset, n_global_syms The variable globals_offset is an integer offset within global _psymbols defined in the structure objfile; n_global_syms is the number of global symbols defined for this source file.

statics_offset, n_static_syms These are similar to the previous fields, but apply to static symbols.

symtab This is a pointer to the symtab structure eventually allocated for this source file. This field is set when the full symbols definition is read and is NULL otherwise.

read_symtab This is a pointer to the function that will read the symbol table information to mutate the partial_symtab into a symtab structure. The name of the function pointed to is xxx_psymtab_to_symtab (GDB-4.X/gdb/xxxread.c), where xxx is the prefix name of the object file format.

read_symtab_private This field contains information used by read_symtab, in order to locate the part of the symbol table that this psymtab corresponds to. This information is specific depending on the format of the object file being read.

readin This field is set to a non-zero value when the symtab structure corresponding to this psymtab has been read in.

The structure section_offsets contains an array of offsets. This information is used to relocate the symbols from each section in a symbol file. The ordering and meaning of the offsets is dependent on the format of the object file.

Struct section_offstes *symtab.h*	
CORE_ADDR	offsets[1]

Figure A.4: The GDB data structure: section_offsets structure

A.3 The Obstack Structure

The next structure described is obstack, which is a stack of objects. This is used in order to manage the memory allocation efficiently. The problem with most of the objects that GDB allocates is that they contain character strings of undefined length.

In practice this often means you will build many short symbols and a few long symbols. At the time you are reading a symbol you don't know how long it is. One traditional method is to read a symbol into a buffer, reallocating the buffer every time you try to read a symbol that is longer than the buffer. Then, this buffer is copied in a more permanent symbol-table entry.

The approach used by GDB using obstacks is more efficient. GDB creates three different obstacks for different kinds of objects, one to store the partial_symbol structures, one to store the symbol structures and one to store the type structures. All symbol names of one type are then allocated on the same obstack. A large chunk of memory is allocated for each obstack. When you want to build a symbol in the chunk, you just add characters starting at next available memory location in the chunk. When you reach the end of the symbol, you know how long the object is, and you can create a new object. Usually, the characters will not exceed the highest address of the chunk, because you would typically expect a chunk to be (say) 100 times as long as an average object.

To refer to the object, we just have to point where it lies. No moving of characters is needed and this is the second advantage of this method: long strings never need to be moved. Once an object is formed, it does not change its address during its lifetime.

When the object exceeds a chunk boundary, we allocate a larger chunk, and then copy the partly formed object from the end of the old chunk to the beginning of the new larger chunk. We then carry on adding characters to the end of the object as we normally would.

The definition of the obstack structure is shown in Figure A.5.

Struct obstack

./include/obstack.h

```
long                            chunk_size
struct  _obstack_chunk          *chunk
char                            *object_base
char                            *next_free
char                            *chunk_limit
int                             temp
int                             alignment_mask
struct  _obstack_chunk*         (*chunkfun)()
void                            (*freefun)()
void                            *area_id
int                             flags
```

Figure A.5: The GDB data structure: obstack structure

The fields in the obstack structure are as follows:

chunk_size This field contains the preferred amount of memory by which we want to increase the obstack.

chunk This is the address of the current _obstack_chunk.

object_base This is the address of the object that we are building.

next_free This represents where to add the next character to the current object.

chunk_limit This is the address just following the current chunk.

temp This is a dummy field.

alignment_mask This is the mask of alignment for each object.

chunkfun This field points to a user's function to allocate a chunk.

freefun This field points to a user's function to free a chunk.

area_id This field selects in which region we want to allocate/free memory.

flags This field contains miscellaneous special purpose flags.

The definition of an _obstack_chunk structure is shown in Figure A.6.

Struct _obstack_chunk
../include/obstack.h

```
char                            *limit
struct  _obstack_chunk          *prev
char                            contents[4]
```

Figure A.6: The GDB data structure: _obstack_chunk structure

The fields in the _obstack_chunk structure are as follows:

limit This is the upper limit of the chunk. This address does not belong to the chunk.

prev This is a pointer to the previous chunk. If there is none, this field is set to null.

contents This is where the objects begin.

A.4 The Partial_Symbol Structure

The objfile structure contains two lists of partial symbols, for global and static symbols. Those lists are implemented using the psymbol_allocation_list structure. The definition of the psymbol_allocation_list structure is shown in Figure A.7.

The fields in the psymbol_allocation_list structure are as follows:

list This is a pointer to an array of symbols.

next This is the address where the next symbol can be created.

Struct psymbol_allocation_list
symfile.h

```
struct partial_symbol  *list
struct partial_symbol  *next
int                    size
```

Figure A.7: The GDB data structure: psymbol_allocation_list structure

size This is the current size in bytes of this list.

The structure partial_symbol, defined in Figure A.8, is a subset of a symbol structure. The main difference is that the type have not been parsed yet for that symbol, and as a consequence, it does not contain the field type of the symbol structure.

Struct partial_symbol
symtab.h

```
char                       *name
enum namespace             namespace
enum address_class         class
union
  {
  long                     value
  CORE_ADDR                address
  }
                           value
```

Figure A.8: The GDB data structure: partial_symbol structure

The fields contained in this structure are the same as the one found in the symbol structure.

A.5 The Minimal_Symbol Structure

The structure minimal_symbol, defined in Figure A.9, holds very basic information about all defined global symbols. The only two required pieces of information are the symbol's name and the address associated with that symbol. In many cases, even if a file was compiled with no special options for debugging, it will contain sufficient information to build a useful minimal symbol table using this structure. Even when a file contains enough debugging information to build a full symbol table, these minimal symbols are still useful for quickly mapping between names and addresses, and vice versa.

Struct minimal_symbol
symtab.h

```
char                        *name
CORE_ADDR                   address
char                        *info
enum minimal_symbol_type
  {
  mst_unknown = 0
  mst_text
  mst_data
  mst_bss
  mst_abs
  }
                            type
```

Figure A.9: The GDB data structure: minimal_symbol structure

The fields in this structure are as follows:

name This required field contains the name of the symbol. The storage for the name is allocated on the symbol_obstack for the associated objfile.

address This required field contains the address of the symbol.

info The info field is available for caching machine-specific information. It is initialized to zero and stays that way until target-dependent code sets it. Storage for any data pointed to by this field should be allocated on the symbol_obstack for the associated objfile. This field is optional.

type This field gives general information about what kind of symbol we are dealing with. This information may not be accurate. The different values that this field may contain are described below.

mst_unknown Unknown type, the default.

mst_text Generally executable instructions.

mst_data Generally initialized data.

mst_bss Generally uninitialized data.

mst_abs Generally absolute (nonrelocatable) data.

A.6 The Sym_Fns Structure

The sym_fns structure has already been presented in Section 4.2.6.

A.7 The Entry_Info Structure

The entry_info structure, shown in Figure A.10, contains information about the entry point, the scope (file/function) containing the entry point, and the scope of the user's main function.

Struct entry_info
objfiles.h

```
CORE_ADDR   entry_point
CORE_ADDR   entry_func_lowpc
CORE_ADDR   entry_func_highpc
CORE_ADDR   entry_file_lowpc
CORE_ADDR   entry_file_highpc
CORE_ADDR   main_func_lowpc
CORE_ADDR   main_func_highpc
```

Figure A.10: The GDB data structure: entry_info structure

The fields of the structure are as follows:

entry_point This is the value corresponding to the entry point for this object file.

entry_func_lowpc, entry_func_highpc These are the addresses of the start (inclusive) and end (exclusive) of the function containing the entry_point.

entry_file_lowpc, entry_file_highpc These are the addresses of the start (inclusive) and end (exclusive) of the object file containing the entry_point.

main_func_lowpc, main_func_highpc These are the addresses of the start (inclusive) and end (exclusive) of the user code main() function.

A.8 The Symtab Structure

The objfile structure contains a pointer to a list of symtab structures, defined in Figure A.11. There is one such structure for each source file that was used in the construction of the object file described by the objfile structure. This structure is the access point for the complete debugging information about the symbols defined in a source file.

Struct symtab

symtab.h

```
enum type_code          code
struct blockvector      *blockvector
struct linetable        *linetable
char                    *filename
char                    *dirname
enum free_code
  {
  free_nothing
  free_contents
  free_linetable
  }
                         free_code
char                     *free_ptr
int                      nlines
int                      *line_charpos
enum language            language
char                     *version
char                     *fullname
struct objfile           *objfile
#if defined (EXTRA_SYMTAB_INFO)
EXTRA_SYMTAB_INFO
#endif
```

Figure A.11: The GDB data structure: symtab structure

The fields of the structure symtab are as follows:

next This field is used to chain the symtabs together.

blockvector This is a pointer to the list of all symbol scope blocks for this symtab.

linetable This is a pointer to a table that maps core addresses to line numbers for this file.

filename This is the name of the source file.

dirname This is the name of the directory in which it was compiled.

free_code This component says how to free the data we point to: free_contents will free everything; free_nothing won't free anything; free_linetable will just free the linetable.

free_ptr This is a pointer to one block of storage to be freed.

nlines This is the total number of lines found in the source file.

line_charpos This is a pointer to an array that maps line numbers to character position.

language This is the language in which this source file was written.

version This field contains the version of the compiler. It may be zero.

fullname This is the full name of the file, as found by searching the source path.

objfile This is a backpointer to the structure objfile, from which this symbol information was read.

The last field is used whenever special debugging information for a target machine can not be represented in a normal symtab.

A.9 The Source and Linetable Structures

The following structures (sourcevector, source, linetable and linetable_entry) are used to describe the relation between source files, line numbers and addresses.

The structure sourcevector, shown in Figure A.12, is the list of source files, and has two fields:

length This is the number of source files described.

source This is an array of source strcutures which contains length elements.

Struct source_vector
symtab.h

```
int                     length
struct source           *source[1]
```

Figure A.12: The GDB data structure: sourcevector structure

The source structure, shown in Figure A.13, contains the information for one source file and has two fields:

name This is the name of the file.

contents This is a pointer to a linetable structure for the mapping between source lines and addresses in memory.

Struct source
symtab.h

```
char                    *name
struct linetable        contents
```

Figure A.13: The GDB data structure: source structure

The linetable structure, shown in Figure A.14, is the list of linetable_entry structures and has two fields:

nitems This corresponds to the number of lines contained in the source file.

item This is an array of linetable_entry's, which contains nitems elements.

Struct linetable
symtab.h

```
int                      nitems
struct linetable_entry   item[1]
```

Figure A.14: The GDB data structure: linetable structure

The linetable_entry structure, shown in Figure A.15, is a mapping between the number of a source line and its address in memory and has consequently two fields:

line This is a line number in the source file.

pc This is an address in memory corresponding to that line.

Struct linetable_entry
symtab.h

```
int                          line
CORE_ADDR                    pc
```

Figure A.15: The GDB data structure: linetable_entry structure

A.10 The Block Structure

All of the name-scope contours of the program are represented by a block structure. All of these structures are pointed to by the blockvector. Each block represents one name scope. The first two blocks in the blockvector are special. The first one contains all the symbols defined in this compilation whose scope is the entire program linked together. The second one contains all the symbols whose scope is the entire compilation excluding other separate compilations. Each block records a range of core addresses for the code that is in the scope of the block. The blocks appear in the blockvector in order of increasing starting-address, and, within that, in order of decreasing ending-address. Figure A.17 and A.16 show the blockvetor and block structure, respectively.

The fields in the blockvector structure are:

nblocks This is the number of blocks in the list.

block This a pointer to the list of block structures.

Struct block
symtab.h

`CORE_ADDR`	`start_address`
`CORE_ADDR`	`end_address`
`struct symbol`	`*function`
`struct block`	`*superblock`
`unsigned char`	`gcc_compile_flag`
`int`	`nsyms`
`struct symbol`	`*sym[1]`

Figure A.16: The GDB data structure: block structure

The fields in a block structure are:

startaddr, endaddr These two addresses give the range of addresses in the executable code that belongs to this block.

function This is a pointer to the symbol that names this block, if the block is the body of a function. Otherwise, this field is set to null.

superblock This is a pointer to a block that would contain this one. An example of such a case is when a function is defined inside a function. If this block is not contained in another one, this field is set to null.

gcc_compile_flag This flag indicates whether the the function corresponding to this block has been compiled with gcc (gcc_compile_flag=1) or not (gcc_compile_flag=0). If this block does not correspond to a function, this block has no meaning.

nsyms This field corresponds to the number of symbols defined in this block.

sym This a pointer to an array of symbol structures, which contains nsyms elements.

Struct blockvector *symtab.h*	
`int`	`nblocks`
`struct block`	`*block[1]`

Figure A.17: The GDB data structure: blockvector structure

A.11 The Symbol Structure

The symbol structure, shown in Figure A.18, contains the full information of a symbol. The fields in this structure are:

name This is the symbol name.

namespace This field contains a code for the type of this symbol. Following is the description of the possible values for namespace.

VAR_NAMESPACE This is the most common namespace. Variables, function names, typedef names and enumerated type values are contained in that namespace.

STRUCT_NAMESPACE This namespace contains all the structures, unions and enumeration types. Thus, if a structure foo is defined in a program, it produces a symbol named foo in this namespace.

LABEL_NAMESPACE This code would be used for names of labels (for unconditional branching). However, this name is currently not used, and labels are not recorded in the data structure created by GDB.

Struct symbol

symtab.h

```
char                          *name
enum namespace                namespace
enum address_class            class
struct type                   *type
unsigned short                line
union
  {
  long                        value
  struct block                *block
  char                        *bytes
  CORE_ADDR                   address
  struct symbol               *chain
  }
                              value
union
  {
  struct
    {
    short                     regno_valid
    short                     regno
    }
                              basereg
  }
                              aux_value
```

Figure A.18: The GDB data structure: symbol structure

UNDEF_NAMESPACE This value is used when we don’t know what it is. This is an error.

class An address-class identifies where to find the value of a symbol. In the case of a symbol having a namespace equal to STRUCT_NAMESPACE, this field is equal to LOC_TYPEDEF, and this symbol has obviously no value.

type This is a pointer to a structure describing the data type of this symbol.

line This is the line number in the source code, where the definition of this symbol has been found.

value This field, used in conjunction with class, gives information about where the value of this symbol is stored.

aux_value This field may contain additional information needed to locate the value.

A.12 The Type Structure

This structure, shown in Figure A.19, contains the type information of the symbols. It is actually referenced by the objfile and by the symbol structure. The objfile structure uses it to define a list of fundamental types and the symbol structure binds each symbol to a particular type.

Struct type

gdbtypes.h

```
enum type_code          code
char                    *name
unsigned                length
struct objfile          *objfile
struct type             *target_type
struct type             *pointer_type
struct type             *reference_type
struct type             *function_type
short                   flags
short                   nfields
struct field
   {
   int                  bitpos
   int                  bitsize
   struct type          *type
   char                 *name
   }
                        *fields
struct type             *vptr_basetype
int                     vptr_fieldno
union type_specific
   {
   struct type          **arg_types
   struct cplus_struct_type
                        *c_plus_stuff
   }
                        type_specific
```

Figure A.19: The GDB data structure: type structure

The fields in the type structure are:

code This code tells us what kind of data type is described in this type structure. Following is the description of the enumeration type_code.

TYPE_CODE_UNDEF This value is used to catch errors.

TYPE_CODE_PTR Pointer type.

TYPE_CODE_ARRAY Array type, lower bound zero.

TYPE_CODE_STRUCT C structure or Pascal record type.

TYPE_CODE_UNION C union or Pascal variant part type.

TYPE_CODE_ENUM Enumeration type.

TYPE_CODE_FUNC Function type.

TYPE_CODE_INT Integer type.

TYPE_CODE_FLT Floating type.

TYPE_CODE_VOID Void type (values have a zero length).

TYPE_CODE_SET Pascal sets type.

TYPE_CODE_RANGE Range type (integers within specified bounds).

TYPE_CODE_PASCAL_ARRAY Array with explicit type of index.

TYPE_CODE_ERROR Unknown type.

TYPE_CODE_MEMBER Member type for C++ classes.

TYPE_CODE_METHOD Method type for C++ classes.

TYPE_CODE_REF C++ Reference types.

TYPE_CODE_CHAR Modula-2 character type.

TYPE_CODE_BOOL Builtin Modula-2 BOOLEAN.

name This is the name of this type. This can be null.

length This represents the length in bytes of storage needed for a value of this type.

objfile This is a backpointer to the objfile structure. Every type is associated with a particular object file and is allocated on the type_obstack for that objfile. This field is used when GDB needs to allocate a new type while it is not in the process of reading symbols from a particular object file. This happens when the type being created is a derived type of an existing type. A pointer to a type is an example of such a situation. So, having a pointer to the objfile structure, we can just allocate the new type using the same objfile type_obstack as the existing type.

target_type This field has a meaning for only some kinds of data types. If the structure defines a pointer to another type, target_type is a pointer to the structure type defining the object pointed to. For an array type, target_type is a pointer to the structure type defining the type of the elements of the array. For a range type, target_type is a pointer to the structure type defining the full range.

pointer_type This field is a pointer to the type structure which defines a type that is a pointer to this type.

reference_type This field is only used for reference to specific C++ types.

function_type In the case of a function, this field is a pointer to a structure type that defines the type of the element returned by the function.

flags This contains miscellaneous flags about this type.

nfields For structure and union types, this field contains the number of fields contained by that structure or union. Note that a C++ class is considered as a structure. For sets and Pascal array types, there is one field whose type is the domain type of the set or array. For range types, there are two fields, the minimum and maximum values. For enumeration types, each possible value is described by one field.

fields This the actual array of fields, as described above, containing nfields elements. Using a pointer to a separate array of fields is useful, because all the types have the same size, and so, we can allocate the space for a type before as we know its size.

bitpos In the case of the description of the fields of a structure, this is the position of this field, counting in bits from the start of the structure. In the case of a function type, this is the position in the argument list of this argument. For a range bound, this is the value itself.

bitsize This field represents the size in bits of this field.

type In the case of a structure or an enumeration type, this is a pointer to the structure type defining the type of this field. In the case of a function type, this is a pointer to the structure type defining the type of this argument. In the case of an array type, this points to the structure type defining the type of the elements of the array.

name This is the name of the field, value, or argument. This field is set to null for range bounds and array domains.

vptr_basetype, vptr_fieldno For classes with virtual functions, vptr_basetype is the base class which defines the virtual function table pointer and vptr_fieldno is the field number of that pointer in the structure. For types that are pointer to member types, vptr_basetype is the type that this pointer is a member of.

type_specific This is a pointer to language-specific information.

arg_types This field is used when the type being described is a function or a method.

cplus_stuff This field points to a cplus_struct_type structure, when the type being described is a C++ class or structure.

A.13 The Cplus_Struct_Type Structure

This data structure, shown in Figure A.20, contains all the information characterizing a C++ class. In particular, the methods, their signature and access control are defined.

The fields in this structure are as follows:

Struct cplus_struct_type

gdbtypes.h

```
B_TYPE                  *virtual_field_bits
B_TYPE                  *private_field_bits
B_TYPE                  *protected_field_bits
short                   nfn_fields
short                   n_baseclasses
int                     nfn_fields_total
struct fn_fieldlist
  {
  char                  *name
  int                 length
  struct fn_field
    {
    struct type   *type
    struct type   **args
    char          *physname
    struct type   *fcontext
    unsigned int  is_const      : 1
    unsigned int  is_volatile   : 1
    unsigned int  is_private    : 1
    unsigned int  is_protected  : 1
    unsigned int  is_stub       : 1
    unsigned int  is_dummy      : 3
    unsigned      voffset       :24
    define VOFFSET_STATIC 1
    }
                  *fn_fields
  }
                        *fn_fieldlists
unsigned char     via_protected
unsigned char     via_public
```

Figure A.20: The GDB data structure: cplus_struct_type structure

virtual_field_bits For derived classes, the number of base classes is given by n_baseclasses and virtual_field_bits is a bit vector containing one bit per base class. If the base class is virtual, the corresponding bit will be set.

private_field_bits For classes with private fields, the number of fields is given by nfields and private_field_bits is a bit vector containing one bit per field. If the field is private, the corresponding bit will be set.

protected_field_bits This field plays the same role as the previous one, except that it applies to protected fields.

nfn_fields This element represents the number of methods defined by this class. It should be noted that functions that are overloaded are counted only once. This field can be seen as the number of methods defined by this class having different names. We don't distinguish methods that have the same name but differ in their signature (i.e. overloaded functions).

n_baseclasses This element represents the number of base classes this class derives from.

nfn_fields_total This is the total number of methods declared for this class, i.e. the number of methods defined by this class plus the methods defined by the base classes it derives from.

fn_fieldlists This is an array of fn_fieldlist containing nfn_fields elements. There is one entry in that array per function name.

- **name** This is the name of the method being described. This name does not include the signature. Apparently, this may not be the full name of a class. For example, if the declaration of a class Y is nested inside the declaration of class X, its name should be X::Y. However, the field name will contain only Y.

- **length** This is the number of methods with this name. This field is necessary because of overloaded functions.

- **fn_fields** fn_fields is a pointer to an array of fn_field which contains length elements. This array contains one entry for each method with this name.

  - **type** This is a pointer to the structure type defining the type of the returned value of the function.

  - **args** This is supposed to be a pointer to list of the types of the arguments of this function. However, this list does not seem to be currently maintained by GDB.

  - **physname** If the method being described is a constructor, the field contains the mangled name of the constructor. From this name, we can then deduce the real name of the class, just by demangling this method name. This is interesting as the field name may not contain the real name of the class, as said before. If the method is not a constructor, then this field contains the signature of the method. As the first argument of a method is always the 'this' pointer, which has the type of this class, we can deduce the name of the class from it. Note that this first argument is a hidden argument: the programmer does not include this argument in the argument list when he writes a pointer, but it is automatically added by the compiler.

fcontext If the method described is a virtual function, fcontext is a pointer to a structure type that describes the base class that first defines this virtual function.

is_const, is_volatile, is_private, is_protected, is_stub These are different flags for some properties of the method being described.

dummy This field is just added to make a one byte word with the previous bits.

voffset This is the index into that base class's virtual function table, increased by 2.

via_protected, via public These fields indicate whether the element described is private, protected or public.